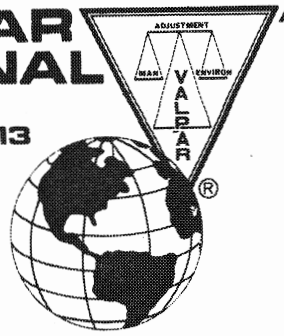


**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

valFORTH 1.1^{T.M.}

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

Software and Documentation
© Copyright 1982
Valpar International

valFORTH^{TM.}
SOFTWARE SYSTEM

valFORTH^{TM.} 1.1

Stephen Maguire
Evan Rosen

(Atari interfaces based on work by Patrick Mullarky)

Software and Documentation
© Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

valFORTH 1.1 USER'S MANUAL

Table of Contents

	<u>Page</u>
I. STROLLING THROUGH valFORTH 1.1	
A brief look at valFORTH 1.1	
a) ERRORS, RECOVERIES, CRASHES	1
b) FORMATTING AND COPYING DISKS	2
c) COLORS	3
d) DEBUGGING	4
e) PRINTING	6
f) EDITING	6
g) GRAPHICS	9
h) SOUNDS	10
i) THE GREAT SCREEN SIZE DEBATE	10
j) SAVING YOUR FAVORITE SYSTEM(S)	11
k) DISTRIBUTING YOUR PROGRAMS	11
II. THE FORTH INTEREST GROUP LINE EDITOR	
The command glossary for the standard fig-FORTH line editor	
III. CREATING DISKS FOR PRODUCTION	
a) RELOCATING THE BUFFERS TO SAVE 2K+	1
b) COMPILING AUTO BOOTING SOFTWARE	3
c) DISTRIBUTING YOUR PROGRAMS	4
IV. valFORTH 1.1 SYSTEM EXTENSIONS	
a) GRAPHIC SUBSYSTEM	1
b) COLORS	3
c) SOUND GENERATION	4
d) TEXT OUTPUT ROUTINES	5
e) DISK FORMATTING AND COPYING	5
f) valFORTH DEBUGGER	6
g) FLOATING POINT PACKAGE	7
h) OPERATING SYSTEM PACKAGE	13
V. valFORTH GLOSSARY	
Descriptions of the entire valFORTH bootup dictionary	
a) fig-FORTH GLOSSARY AND valFORTH EXTENSIONS	1
b) valFORTH MEMORY MAP	28
VI. valFORTH ADVANCED 6502 ASSEMBLER	
A user's manual for the valFORTH assembler.	
VII. valFORTH 1.1 SUPPLIED SOURCE LISTING	

STROLLING THROUGH valFORTH 1.1

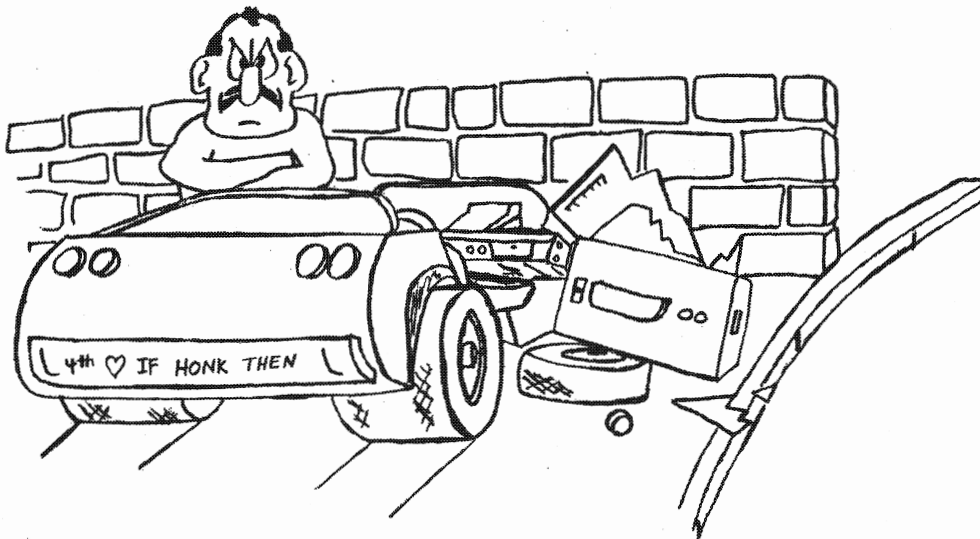
Welcome. For this excursion you'll need an ATARI 800 (or 400) with at least 24K, a disk drive, monitor, a printer, and valFORTH 1.1. You could even do without the printer. Please get everything up and running, and boot valFORTH.

(To boot the disk, turn the drive(s) on and the computer off. Insert the disk in drive 1 and turn the computer on. The disk should now be booting, and the monitor speaker should be going beep-beep-beep-beep as valFORTH loads.)

ERRORS, RECOVERIES, CRASHES

Before we get started, let's mention the inevitable: Most of the time when you make an error you'll receive one of the fairly lucid fig-Forth error messages. If you just get a number, this will probably refer to the Atari error message list which you can find in the documentation that came with your computer. Since the Atari is a rather complex beast, you may sometimes get into a tangle that looks worse than it is. Keep your head. If you have party-color trash on the screen, for instance, and yet you can still hear the "peek-peek-peek" of the key when you hit return, you may have merely blown the display list without hurting your system. Try Shift-Clear followed by 0 GR. . Very often you're home again. If this doesn't work, try a warm start: Hold down a CONSOLE button, say START, and while you've got it down, press SYSTEM RESET and hold both for a moment until the "valFORTH" title comes up. (If you were to push the SYSTEM RESET button alone, you'd get a cold start, which takes you back to just the protected dictionary.) A warm start gets you back to the "ok" prompt without forgetting your dictionary additions. If warm start doesn't work, your system is being kept alive only by those wires connected to it; it no longer has a life of its own. The standard procedure now is to push SYSTEM RESET alone a few times (cold start) in a superstitious manner, and then reboot the system.

Look carefully at the code that blew the system last time. If you're really having trouble debugging, sprinkle a bunch of WAIT's and/or .S's (Stack Printouts) through the code, and go through again. The best thing about those first few long debugging sessions in any computer language is that they teach you the value of writing code carefully.



FORMATTING AND COPYING DISKS

You may have noticed that your system came up in a green screen. In a little while you'll be able to change it to anything you like. We'll get to that in a moment, but right now type 170 LIST (and then hit RETURN.) Behold the table of contents. Our first priority should be to make a working disk by copying the original.

Let's assume that you have a blank, unformatted disk on which to make your copy. Notice the line called FORMATTER on screen 170. At the right side of this line is probably 92 LOAD, though the number may be different in later releases. Type 92 LOAD (or whatever the number is) and wait until the machine comes back with "ok". Now you're going to type FORMAT, but for safety's sake why not remove the valFORTH disk and insert the blank disk? One never knows if newly purchased software will give you warnings before taking action. ("Warnings" or "Prompts" make a system more friendly.) Ok, now type FORMAT. For the drive number you probably want to hit "1", unless you've got more than one drive and don't want to format on the lowest. In answer to the next prompt, hit RETURN unless you've changed your mind. Now wait while the machine does the job. If you get back "Format OK" you're in business. (If "Format Error" comes back, suspect a bad blank disk or drive.) You might as well format another disk at this time on which to store your programs.

Now to make the copy. Return the valFORTH disk to the drive and do 170 LIST again. Find DISK COPIERS and do 72 LOAD, or whatever number is indicated. When the "ok" prompt comes back, two different disk copying routines are loaded: DISKCOPY1 for single drive systems and DISKCOPY2 for multiple drive systems. Type whichever of these words is appropriate and follow the instructions. ("source" means the disk you want to copy. "dest." is the blank "destination" disk.) There are 720 sectors that have to be copied. Since this can't be done in one pass, if you are using DISKCOPY1 you will have to swap the disks back and forth until you're done. (The computer will tell you when.) The less memory you have, the more passes; there is great benefit in having 48K. If you have more than one drive, it still takes several internal passes, but there is no swapping required. Either way, the process takes several minutes with standard Atari disk drives.



Nice going. Now store the original disk in some safe place. Don't write protect your copy yet. First we'll adjust the screen color to your taste. Just to see if you really have a good copy, boot it. This can be done by the usual on-off method, or by typing BOOT.

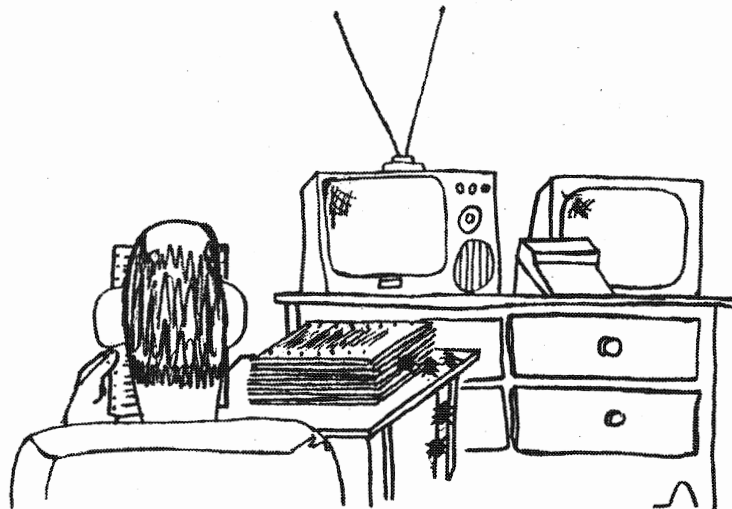
COLORS

Before playing with the colors, let's look at something else. Type VLIST, and watch the words go by. These are all of the commands that are currently in the "dictionary" in memory in your system. You can cause this listing, or any other, to pause by hitting CTRL and 1 at the same time. This is a handy feature of the Atari. The listing is restarted with the next CTRL 1. Additionally, in valFORTH most listings may be aborted by pressing any of the three yellow buttons START, SELECT, and OPTION. These three buttons together will be referred to as the CONSOLE.

Do VLIST again, and abort it with a CONSOLE press after a few lines. At the top of the list you should see the word TASK. Remember that for a moment. Do 170 LIST again. Look over the list and find COLOR COMMANDS, and LOAD as appropriate. Now do VLIST again, and stop the list when it takes up about half of the screen. Above TASK you now can see a number of new commands, or "words" as they are commonly called in Forth. These were added to the dictionary by the LOAD command. Here's what some of these words do:

Type BLUE 12 BOOTCOLOR, and you get a new display color. Try BLUE 2 BOOTCOLOR. If you try this action with the number as 4, the letters will disappear, and you'll have to type carefully to get them back. The number is the luminance or "lum" and is an even number in the range 0 to 14. The color-name is called the "hue." The word "color" will be used to refer to a particular combination of hue and lum; hence PINK 6 is a color, PINK is a hue. There are 16 hues available and you can read their names from the display, starting with LTORNG ("light orange") and ending with GREY. (The hues may not match their names on your monitor. Later on you'll be able to change the names to your liking, or eliminate them altogether to save memory, and just use numbers. For instance, PINK is equal to 4.)

Try out different colors using BOOTCOLOR, until you find one you can live with for a while. We usually use GREEN 10 or GREEN 12 in-house at Valpar. While you are doing this you'll probably make at least one mistake, and the machine will reply with an error message like "stack empty." Just hit return to get the "ok" back and start whatever you were doing again. Actually, you don't even have to get the "ok" back, but it's reassuring to see it there. When you've got a color you like, do VLIST again. Note the first word above TASK. It should be GREY. Carefully type FORGET GREY, and do VLIST again. Notice that GREY and all words above it are indeed forgotten. That's just what we want. Now type SAVE. You'll get a (Y/N) prompt back to give you a chance to change your mind, since SAVE involves a significant amount of writing to drive #1. For practice, check to see that you still have the copy in drive one, and if it is there, hit Y, and off we go. When "ok" comes back, remove the disk and apply a write protect tab to it. Boot this disk again to see that it will come up in your selected color.



DEBUGGING

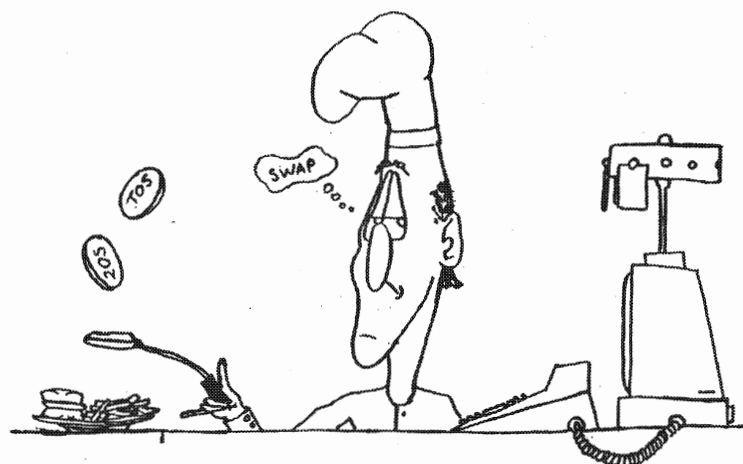
Look at 170 again. Load the DEBUGGING AIDS, and type ON STACK. You'll see that the stack is empty. Great, what's a stack? The best answer to this is to suggest that you read Leo Brodie's book, Starting FORTH. This amusing and thorough treatment of FORTH starts from the novice level and continues on to most of the advanced concepts in FORTH. Starting FORTH is available from many sources, including Valpar International. Included with your valFORTH package is a document called "Notes on Starting FORTH for the fig-FORTH User" which pinpoints differences between Brodie's dialect of FORTH, called 79 Standard, and the somewhat more common fig-FORTH on which valFORTH was based. It is not feasible to present a course on FORTH in these pages, since FORTH is far more powerful than BASIC, which itself requires a fair amount of space to present. However, we'll try to be as considerate as possible to the FORTH-innocent.

The visible stack is a very good debugging and practice tool. Type a few integers, say 5 324 -19 0 and hit RETURN. The numbers are now visible on the stack. Top of stack, TOS, is at right. Print the top entry by typing "." and then do DUP. Note that there are now two -19's on top. Do "*" to multiply them together. Now do DROP to discard the product, 361, currently at TOS. Ok, now do SWAP to exchange the 324 and 5 and then do "/" to divide 324 by 5. This should leave 64, since the answer is truncated. Now type 1000 * and notice that instead of getting 64000 you get -1536. This is of course two's complement on a two-byte number. Type U.S which will switch the visible stack to unsigned representation. Type .S to go back. The words .S and U.S may be used with the visible stack on or off to show the stack one time. Now type OFF STACK. Type in a few more numbers, say 1 2 3 4 5 6 7. ON STACK again, and observe that the entries are retained. Do OVER to bring a copy of the 6 over the 7. Now DROP it. Do ROT to rotate the third from top, 5, to the top. Now do <ROT to put it back. In addition to all of these normal routines, valFORTH supports PICK and ROLL both coded in 6502 for speed. Notice that the 5th on stack is a 3. 5 PICK will bring a copy of it to TOS. Do this and then DROP the 3. Do 5 ROLL to pull the 3 out of the stack and place it at TOS. DO SP! to clear the stack.

One point about number bases: Right now you're in DECIMAL. By typing HEX you go into hexadecimal, and typing DECIMAL or its abbreviation DCX you get back. And, as usual, virtually any base may be used by typing N BASE ! where N is the base you want. Thus, 2 BASE ! gives binary, etc. Some errors, particularly during loading, may leave you in an unexpected base, like base 0, for instance. If you find the machine acting normally except for numbers, this may have happened. A simple DCX will get you back to decimal. The word B? will print the current base in decimal. Put 30 on the stack and then do HEX. Now do B?. Do DCX to return to decimal.

While we're on the subject of numbers, do ON and note that it is just a CONSTANT equal to 1. Similarly, do OFF and see that it is zero. Try 0 STACK and then 1 STACK. The words ON and OFF are provided to enhance readability of code, but could be substituted by 1 and 0 if desired. The two representations are equally fast.

We mention to the newcomer to FORTH that the stack takes the place of dummy variables or dummy parameters in other languages. This reduces memory overhead in several ways but does exact a penalty of reduced readability of FORTH source code. Consistent and sensible source code formatting can significantly enhance readability. The source code on the present disk may be used as a reasonably good example of well-arranged code.



Now a few words about DECOMP. Clear the stack. Type in 3 and 4. Do OVER OVER followed by 2DUP and notice that these two phrases have the same effect. Clear the stack and then turn it off if you like, and do DECOMP 2DUP. What you see is a decompilation of 2DUP which indicates that it is indeed defined as OVER OVER. Decomp OVER. The word "primitive" in the decompilation of OVER indicates that OVER is defined in machine code.

Decomp LITERAL. The word (IMMEDIATE) after LITERAL in the decompilation indicates that LITERAL is immediate. Not all words can be decompiled by DECOMP, and sometimes trash will be printed with long pauses between lines. In this case, hold down any CONSOLE button (the three yellow ones, remember) until the "ok" comes back. This may take several seconds, but rarely much longer.

PRINTING

If you have a printer attached, we can generate some hardcopy. Look at screen 170 again. You can see the line labeled PRINTER UTILITIES. Don't load it, though. The printer utilities were loaded automatically when you loaded the debugging aids, and so are in the dictionary already. (There is no need to have them in twice, though it wouldn't hurt.) You have access to the words P:, S:, LISTS, PLISTS, PLIST, and a couple of others relating to output. Do VLIST and see if you can spot this group. As a matter of fact, do ON P: VLIST OFF P: all in one shot. ON P: is used to route output to the printer or not. OFF P: stops sending to the printer. Try ON P: OFF S: 170 LIST CR OFF P: ON S: and notice that this time text is not sent to the display screen, only to the printer. That's because of OFF S: .

Look at screen 170 again, either on display or in hardcopy, and note which screen the printer utilities start. Type this number in, but don't type load. Instead, after the number, type 10 PLISTS. This prints 10 screens starting from the first screen you just typed in. If you have a reasonably smart printer, it will automatically paginate, so that the screens are printed three to a page. If the printer acts peculiarly after printing each third screen, the pagination code in the word EJECT is probably not right for your printer. You'll be able to change this later on.

Now type 30 150 LISTS and after a few blank screens you'll see the entire disk go by, except for the boot code. You can pause any time by CTRL 1 or stop by holding a CONSOLE button.

Finally, do ON P: 30 179 INDEX OFF P: to print a disk index. The index is made up of the first line of each screen.



EDITING

Two editors have been included in this package. The fig (Forth Interest Group) Editor and the valFORTH 1.0 Editor. The latter, while a perfectly useable video-display editor in its own right, is actually a stripped-down version of the valFORTH 1.1 Editor, available with the Utilities/Editor package from Valpar International. The 1.0 Editor is provided to give the user some idea of what the very powerful 1.1 Editor is like, without actually providing it. (Among other things, the 1.1 Editor has a user-definable line buffer of up to 320 lines with a 5 line visible window at the bottom of the display. This window can be seen at the bottom of the 1.0 Editor, but is inactive.)

The fig Editor is a general-purpose FORTH line editor, and was the FORTH editing workhorse until good video-displays were developed.

The fig Editor User Manual is located just after this section. It is based on that by Bill Stoddart of FIG, United Kingdom, published in the fig-Forth installation manual 10/80, and is provided through the courtesy of the FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070. Serious Forth programmers should write FIG to request their catalog sheet of references and publications.

Let's look at the valFORTH editor 1.0. Refer to the directory again, screen 170, and load the valFORTH editor. (Don't load the fig Editor by mistake.) Before proceeding, make sure that the write-protect tab on your disk is secure. The word to enter the editor at the screen on top of stack is V. You can remember it by thinking of it as "view." Type 170 V. Screen 170 is now on the display again, but in the valFORTH 1.0 Editor rather than as a listing. This Editor is a subset of the valFORTH 1.1 Editor available in the Editor/Utilities package, which is MUCH more powerful and convenient, and is priced far lower than any comparable product of which we are aware. The Editor Command card provided shows all of the commands available with the 1.1 Editor. Commands available with the 1.0 Editor are marked with asterisks (*) on the card. Let's run through them:

The cursor can be moved as in the Atari "MEMO PAD" mode. That is, hold down the control key (CTRL) and move the cursor around the display with the four arrow keys. To enter text (replace mode only in 1.0), position the cursor and type it in. Delete characters with the backspace key as usual. The cursor will wrap to the next line at the end of a line, and to the top of the screen when it goes off the bottom. You can type at will on this screen since we won't save the changes to disk.

Do a Shift-Insert and notice that a blank line is inserted at the cursor line. The bottom line is lost, though it is recoverable in the 1.1 version. Now do Shift Delete to remove a line. (Delete is on the Backspace key). These are all of the Editing commands available in the 1.0 Editor. There are two methods of exiting the editor, CTRL S and CTRL Q. CTRL S marks the screen for saving to disk, and CTRL Q forgets the latest set of editing changes. As usual, changes are not saved immediately. This is accomplished with the word FLUSH or by bringing other screens into the buffers and pushing the edited ones out. Again, as usual, the EMPTY-BUFFERS command, or its valFORTH abbreviation, MTB, will clear all buffers, thus forgetting any changes that have not yet been written to disk.

Try CTRL Q to exit now. Reedit the screen by typing L. L does not require an argument on stack and will bring the last-editing screen into the editor. The words CLEAR and COPY have their normal meanings, as does WHERE, which has had the standard fig bug fixed. See the glossary for details. Note that since COPY in valFORTH does not FLUSH its changes, careful use allows transfers of single screens between disks by swapping disks after COPY and before FLUSH. This is particularly handy, for example, for transferring error message screens 176-179 between disks.

You can make this transfer by doing

```
176 176 COPY 177 177 COPY 178 178 COPY 179 179 COPY
```

and then swapping in the destination disk and typing FLUSH. You may want to define a word to do this automatically:

```
: ERRXFR      ( -- )  
  CR ." Insert source and press START" WAIT  
  180 176  
  DO I I COPY  
  LOOP  
  CR ." Insert dest. and press START" WAIT  
  FLUSH ;
```

Because there are four 512 character screen buffers in memory in valFORTH, four 512 characters screens at a time is the maximum for this method. Bulk screen moves on a single disk or between disks are available with the Utilities/Editor Package.

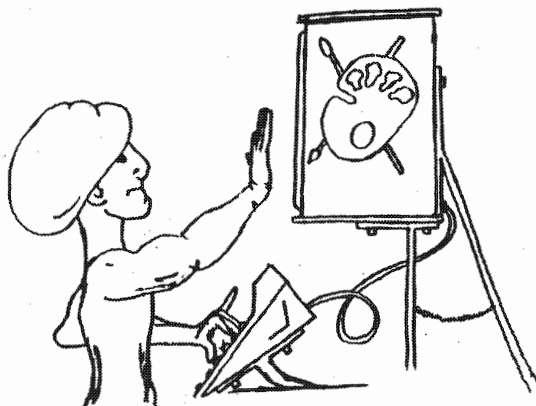
Note: The word "screen" in Forth refers to an area of the disk. When you do 170 LIST you are listing screen 170. In valFORTH there are 180 screens, numbered 0-179, on the disk in drive 1. In multiple-drive systems screen numbers continue across drives, so that screens 180-349 are on drive 2. 180 LIST will automatically read from drive 2. For technical reasons screen 0 should not be used for program code.

Whichever editor you use for the moment, you can write your programs to a blank disk and load them from there. Remember that in fig-FORTH (and so also in valFORTH), if you wish to continue loading from one screen to the next, all but the last screen should end in -->. You'll see this all through the valFORTH 1.1 code. You'll also see ==>. For present purposes you can use --> everywhere, and forget about ==>. ==> is actually a "smart" version of --> that does nothing if the system uses 1024 character screens instead of 512.

If you are a FORTHER, and wish to use 1024 byte screens, do FULLK. To return to 512 character screens, do HALFK. (A working disk may be SAVE'd in either condition.) Note that the valFORTH 1.0 Editor will not edit 1024 character screens, though the 1.1 version will, and includes special 1K notation. In the same vein, the word KLOAD that appears in the source code is a smart load. See the Glossary for details.

To terminate loading one simply omits the --> on the last screen. ;S may be used to end loading at any point. Also note that valFORTH --> and ==> are smart in the sense that if you wish to stop loading before the machine is ready to stop, simply hold down a CONSOLE button. When --> or ==> execute, they first check the CONSOLE. If a button is pressed, they stop loading instead of continuing with the next screen.

Before leaving editing practice, type MTB to empty the disk buffers and assure yourself that nothing will be flushed to disk accidentally as you read in new screens. Or else, do FLUSH if you really want to save your changes. (Remember to remove the write-protect tab if you do.)



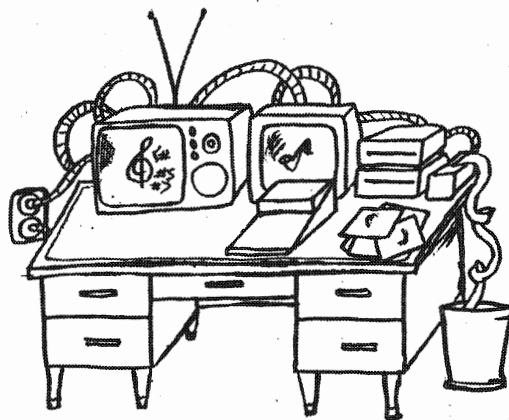
GRAPHICS

On to Graphics. Check screen 170 and load the Color Commands again, and then the Graphics Package. VLIST to see what you've got, and print the list if you like. You may notice that GR. is not among these freshly loaded words: It is in the kernel, that is, the booted code. Try the following sequence:

```
2 GR. (BASIC Graphics mode 2)
5 5 POS. (Move the graphics cursor)
G" TEST" (Send text to graphics area)
1 COLOR (Pick a new graphics color)
G" TEST" (More text)
: SMPL 4 0 DO I COLOR G" TEST" LOOP ; (automate)
SMPL SMPL SMPL (Try it out)
: MANY BEGIN SMPL ?TERMINAL UNTIL ; (More automation)
1 GR. (Go somewhere else)
MANY (Press CONSOLE button to exit)
17 GR. MANY (Try it in full screen)
2 GR. 2 PINK 8 SE. MANY (Use SE. to change color 2)
4 GOLD 8 SE. (Use SE. to change background color).
0 GR. (Go back to normal text screen.)
```

You can also see a quick demonstration by loading the Graphics Demo Program listed on screen 170. If it's not listed on screen 170, do an INDEX in the area of the Graphics routine screens you loaded recently. When you find the Graphics examples screen, load it. Then do FBOX. Take a look at the code and then at the Glossary to get the idea.

As in Atari Basic, adding 16 to the graphics mode you want to enter gives non-split screen, and adding 32 suppresses erase-on-setup of the mode.



SOUNDS

As a final stop on this tour, load the SOUNDS words. The word SOUND acts similarly to the Basic command SOUND. In valFORTH it also has the abbreviation SO. and expects stack arguments like so:

```
channel(0-3) frequency#(0-255) distortion(0-14 evens) volume (0-15).  
    (We use "CatFish Don't Vote" as a mnemonic).
```

Try, for instance 0 200 12 8 SO. and then turn it off with 0 XSND which just shuts off the indicated voice, 0, or XSND4 which quiets everything. More about sound generation by the Atari may be found in the "sound" section.

Logical Line Input

One of the nice features of the Atari OS is that it lets you back the cursor over code that you've typed in already, even edit it with various inserts, deletes, and retypes, and then hit return to have it reinterpreted. This function is supported by valFORTH, and you can re-input up to two full lines of text, (and a wee bit more) at a time just by moving the cursor onto the "logical line" you wish to re-read. Try it.

THE GREAT SCREEN SIZE DEBATE

The "standard" Forth screen is composed of 1024 bytes. This is a nice round number, and on a good text display one can have room for that many characters plus a few more. However, beyond tradition, there is very little functional reason to have 1024 byte screens over several other power-of-2 sizes. In the case of Atari and Apple machines, 512 byte screens make video display editors much easier to work with, since one can get a whole screen in the display at once. valFORTH supports both 1024 and 512 byte screen modes, but in-house at Valpar we strongly prefer 512 byte screens and recommend that you adopt this as your personal standard. If at any time you wish to change to 1K to help compile software written on 1K screens, you can do so with one word, FULLK.

SAVING YOUR FAVORITE SYSTEM(S)

Well, you've seen many of the bells and whistles of valFORTH. When you are using the language for software development you will probably have a favorite set of capabilities that you always want aboard. Rather than loading them from scratch each time, why not SAVE them to a formatted disk? Just get everything you want into the dictionary. After it's all loaded, put a formatted disk into drive 1 and type SAVE. Answer the prompt by pressing "Y" unless you have changed your mind, and the computer will save a bootable copy of your system dictionary on the blank disk.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

- (1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "compiling Auto-Booting Software" section of this manual.
- (2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

Created in whole or part using valFORTH products of
Valpar International, Tucson, AZ 85713, USA
Based on fig-FORTH, provided through the courtesy of
Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

Hope you've enjoyed the tour. Bye now.

FIG EDITOR USER MANUAL

Based on the Manual
by Bill Stoddart
of FIG, United Kingdom

valFORTH organizes its mass storage into "screens" of 512 characters, with the option of 1024. If, for example, a diskette of 90K byte capacity is used entirely for storing text, it will appear to the user as 180 screens numbered 0 to 179. Screen 0 should not be used for program code. Each screen is organized as 16 lines with 32 characters per line.

Selecting a Screen and Input of Text

To start an editing session the user types EDITOR to invoke the appropriate vocabulary.

The screen to be edited is then selected, using either:

n LIST (list screen n and select it for editing) OR
n CLEAR (clear screen n and select for editing)

To input new text to screen n after LIST or CLEAR the P (put) command is used.

Example:

0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, AND 2 OF THE SELECTED SCREEN.

Based on material provided through the courtesy of the FORTH INTEREST GROUP,
P.O. Box 1105, San Carlos, CA 94070.

Line Editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text used by or saved with a line editing command, or a text string to be found or deleted by a string editing command.

PAD can be used to transfer a line from one screen to another, as well as to perform edit operations within a single screen.

Line Editor Commands

- n H Hold line n at PAD (used by system more often than by user).
- N D Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up 1 line.
- n T Type line n and save it in PAD.
- n R Replace line n with the text in PAD.
- n I Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.
- n E Erase line n with blanks.
- n S Spread at line n. n and subsequent lines move down 1 line. Line n becomes blank. Line 15 is lost.

Cursor Control and String Editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided for the user to position the cursor, either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

Commands to Position the Cursor

- TOP Position the cursor at the start of the screen.
- N M Move the cursor by a signed amount n and print the cursor line.
The position of the cursor on its line is shown by a ● (solid circle).

String Editing Commands

- F text Search forward from the current cursor position until string "text" is found. The cursor is left at the end of the text string, and the cursor line is printed. If the string is not found an error message is given and the cursor is repositioned at the top of screen.
- B Used after F to back up the cursor by the length of the most recent text.
- N Find the next occurrence of the string found by an F command.
- X text Find and delete the string "text."
- C text Copy in text to the cursor line at the cursor position.
- TILL text Delete on the cursor line from the cursor till the end of the text string "text."
- NOTE: Typing C with no text will copy a null (represented by a heart) into the text at the cursor position. This will abruptly stop later compiling! To delete this error type TOP X 'return'.

Screen Editing Commands

- n LIST List screen n and select it for editing
- n CLEAR Clear screen n with blanks and select it for editing
- n1 n2 COPY Copy screen n1 to screen n2.
- L List the current screen. The cursor line is relisted after the screen listing, to show the cursor position.
- FLUSH Used at the end of an editing session to ensure that all entries and updates of text have been transferred to disc.

Editor Glossary

TEXT c ---

Accept following text to pad. c is text delimiter.

LINE n --- addr

Leave address of line n of current screen. This address will be in the disc buffer area.

WHERE n1 n2 ---

n2 is the block no., n1 is offset into block. If an error is found in the source when loading from disc, the recovery routine ERROR leaves these values on the stack to help the user locate the error. WHERE uses these to print the screen and line nos. and a picture of where the error occurred.

R# --- addr

A user variable which contains the offset of the editing cursor from the start of the screen.

#LOCATE --- n1 n2

From the cursor position determine the line-no n2 and the offset into the line n1.

#LEAD --- line-address offset-to-cursor

#LAG --- cursor-address count-after-cursor-till-EOL

-MOVE addr line-no ---

Move a line of text from addr to line of current screen.

H n ---

Hold numbered line at PAD.

E n ---

Erase line n with blanks.

S n ---

Spread. Lines n and following move down. n becomes blank.

D n ---

Delete line n, but hold in pad.

M n ---

Move cursor by a signed amount and print its line.

T n ---

Type line n and save in PAD.

L ---

List the current screen.

R n ---
 Replace line n with the text in PAD.

 n ---
 Put the following text on line n.

I n ---
 Spread at line n and insert text from PAD.

TOP ---
 Position editing cursor at top of screen.

CLEAR n ---
 Clear screen n, can be used to select screen n for editing.

FLUSH ---
 Write all updated buffers to disc.

COPY n1 n2 ---
 Copy screen n1 to screen n2.

-TEXT Addr 1 count Addr 2 -- boolean
 True if strings exactly match.

MATCH cursor-addr bytes-left-till-EOL str-addr str-count
 --- tf cursor-advance-till-end-of-matching-text
 --- ff bytes-left-till-EOL
 Match the string at str-addr with all strings on the cursor line
 forward from the cursor. The arguments left allow the cursor R# to
 be updated either to the end of the matching text or to the start of the
 next line.

1LINE --- f
 Scan the cursor line for a match to PAD text. Return flag and update
 the cursor R# to the end of matching text, or to the start of the
 next line if no match is found.

FIND ---
 Search for a match to the string at PAD, from the cursor position
 till the end of screen. If no match found issue an error message
 and reposition the cursor at the top of screen.

DELETE n ---
 Delete n characters prior to the cursor.

N ---
 Find next occurrence of PAD text.

F ---
 Input following text to PAD and search for match from cursor position
 till end of screen.

B ---
Backup cursor by text in PAD.

X ---
Delete next occurrence of following text.

TILL ---
Delete on cursor line from cursor to end of the following text.

C ---
Spread at cursor and copy the following text into the cursor line.

RELOCATING BUFFERS

The purpose of this section is to show you how to avoid incorporating buffer space into an auto-booting program, thereby saving more than 2K in memory requirement for the machine on which the program will eventually run.

Fig-FORTH (and so valFORTH) uses a virtual memory arrangement which allows disk areas to be accessed in a manner similar to that used to access semiconductor memory. We won't go into detail here; those wishing to find out more about this can contact FIG for documentation at:

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

or they can puzzle out the process by starting at the word BLOCK. For our present purposes, however, we simply note that the virtual memory scheme requires that some continuous area of memory be allotted as buffer space for disk operation. valFORTH as delivered has buffer space for four 512 byte "screens" at a time. Each screen is composed of four blocks of 132 bytes each: 128 bytes of actual data, corresponding to a sector, and four bytes of identification and delimiting data. This produces a total of $4 \times 4 \times 132 = 2112$ bytes that are needed for programming and compilation* but are generally not required when software is actually run. In order to get the full use of your computer, particularly for the purposes of producing auto-booting software like games, you'll need to know how memory is mapped and what changes you can make in the mapping. During the following discussion refer to the memory map provided with your documentation.

You will note from the memory map that the buffers are placed just above the kernel (boot-up) valFORTH dictionary. The dictionary pointer is set just past the buffers, so new word definitions will be compiled in above the end of the buffers. Why such an odd location? Read on...

* Those used to seeing the buffers at the top of memory will quickly realize that this is impractical on the Atari, since that area is used for display lists. Although it is possible to an extent to fool the operating system into thinking that it has less memory than it actually has, and thus "reserve" an area at the top of memory, this is a troublesome proposition.

* Another approach is to put the buffers just below the kernel dictionary, which has been done in at least one FORTH-for-Atari release. While this is safe, it sacrifices 2K bytes during run time unless rather clever programming techniques are used on each program to put code into the dormant buffer area.

* Clearly, the buffers should be put somewhere above the dictionary but below the display-list area, and a simple means to relocate them should be supported. This is precisely what you have in valFORTH.

* In a pinch, you can compile using only 264 bytes of buffer memory.

When you have a program that will compile and run, preferably without errors, and you'd like to create a smaller auto-booting version, follow this procedure:

- * Boot the valFORTH disk.

- * Decide on the area to which to relocate the buffers: If the program can be loaded without leaving the 0 Graphics mode or doing anything else to high memory while loading, then the result printed by the sequence

0 GR. DCX 741 @ 2113 - U (See note below.)

will be a safe place to put the buffers; 741 @ is the Atari OS pointer to just below the current display list. (If you will be using Transients, a capability of the Utilities/Editor package, their default location is

DCX 741 @ 4000 -

so you would be better off to put the buffers at, say,

DCX 741 @ 6113 -

to avoid conflict).

- * Find the buffer relocation utility listed in the table of contents starting on screen 170 of the valFORTH disk, and load it. This is a self-prompting utility that directs you to relocate the buffers and then forget the utility. Follow the directions. You'll receive a verification message after the buffers have been moved.

- * Type

' TASK DP !

to move the dictionary pointer below the old buffer area. (Advanced programmers: This is not a typo. The cfa of TASK points to NEXT.)

- * Now load your program as usual. You should probably create an auto-booting program at this point, rather than doing anything else, since if you run the program now it may write into your relocated buffers and conceivably even attempt a write to your disk. So, create an auto-booting version as directed in the Auto-booting section above. Remember that if the program is for distribution, you MUST protect your software and ours by using the AUTO command.

*****CAUTION*****

The buffers start out just above the kernel dictionary, as indicated, and for normal programming they should be LEFT THERE: Several routines on the valFORTH disk and other disks in this product line use the area between pad and the bottom of the display list as a scratch area for extensive disk transfers. DISKCOPY1 and DISKCOPY2 on the valFORTH disk are examples.

Note: The buffers should generally be relocated to an even address because of an Atari OS bug. See also Note 1 at end of valFORTH 1.1 Glossary.

COMPILING AUTO-BOOTING SOFTWARE

Your purchase of valFORTH and its associated packages also grants you a single-user license for the software. You may not copy valFORTH or its associated Valpar International products for any purpose other than for your own use as back-up copies. However, a word called AUTO has been provided to allow you to create a copy of your software that is suitable for distribution. The word AUTO does several things.

* AUTO provides extensive protection both for your software and the valFORTH and auxiliary programs on which it is based. Your product may still be copied by normal methods, but the programming concepts on which it is based will be very difficult to analyze. The valFORTH and auxiliary programs will be rendered useless except to run your program. Since AUTO scrambles all headers in the code before saving to disk or cassette, even direct examination of the code on the medium is not very revealing. This provides essentially all the protection of headerless code.

* AUTO will create a disk that autoboots to the FORTH word of your choice. This usually will be the last word defined in your program. In addition, a disk created using AUTO will not have exit points: That is, even if your program terminates, or makes an error because of an undiscovered bug, it will not exit to valFORTH and the "ok" prompt. Instead, it will automatically attempt to start again at the original auto-boot word, and will do so unless an error has disabled the system.

* AUTO allows repetitive saving of your protected software to disk and cassette in one sitting, with extensive prompting. This provides a short-run production environment. (Remember that if you want to save to cassette, the cassette recorder should be attached to the system at boot time; if it is attached after booting, the computer may not know that the recorder is there and may fail when trying to AUTO to cassette).

To run AUTO and create your bootable software:

- (1) Load valFORTH.
- (2) Relocate buffers to save 2K+, if desired (see below).
- (3) Load your program.
- (4) DISPOSE transients, if you use them. (The Transient utilities come with the Utilities/Editor package, and allow use of "disposable assemblers" and the like).
- (5) Find the Auto-Boot Utility section on the valFORTH disk by referring to the directory starting on screen 170, and load as indicated.
- (6) Type AUTO cccc where "cccc" is the word which you wish to execute on auto-booting the software. You will now be prompted through the rest of the procedure. On exiting from AUTO you will fall through to the auto-booting program that you have just protected.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

- (1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "Compiling Auto-Booting Software" section of this manual.
- (2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

Created in whole or part using valFORTH products of
Valpar International, Tucson, AZ 85713, USA
Based on fig-FORTH, provided through the courtesy of
Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

be included either on the outside of the media (diskette, cassette, or other) as distributed, or in the documentation provided with the product. Please note that failure to include this message with products that include valFORTH code may be regarded as a copyright violation.

GRAPHICS, COLORS, AND SOUNDS

Graphics

The Graphics package follows the Atari BASIC graphics set as closely as possible, and is identical in most respects. As in BASIC, the most complex parts of Graphics are DRAWTO (abbreviated "DR.") and FIL, and even these are not too obscure. Find the Graphics Demo by looking at the directory starting on screen 170, and load it. Try the word FBOX. Now look at the code that produced this effect, if you like. The general explanation is as follows:

Display positions are denoted by two coordinates, a horizontal and a vertical. The 0,0 point is in the upper left hand corner, and the vertical coordinate increases as you go down the display, while the horizontal coordinate increases as you go to the right. This is all familiar from BASIC.

In graphics modes, a single point at position X Y can be plotted by X Y PLOT. The color of the point will be that in the color register declared by the last COLOR command. A line, again of the color in the register declared by the last color command, may then be drawn to point X1 Y1 by X1 Y1 DR. . The word FIL may be used to fill in an area as described in the Atari manual, and as illustrated in the FBOX example. The color register for the fill is the one whose number is on the stack when FIL is executed. Essentially, to set up FIL you draw in boundaries and pick two points you wish to FIL between. The first of these points is set up either by a DR. or PLOT command, or by valFORTH's POSIT command. POSIT has the advantage of not requiring that you put anything into the place where you are positioning yourself. The second point for the FIL command is then set up by using POS. . The fill is then performed by putting a number on stack (the color register for the fill) and then doing FIL.

If you are in a text mode, a single character, c , can be sent to the display by ASCII c CPUT. Text strings can be sent to the display with G" cccc " and in addition will have the color in the register specified by the last COLOR command before the string is output. This is a significant enhancement to BASIC.

Graphics and Color Glossary:

- SETCOLOR** n1 n2 n3 --
Color register n1 (0...3 and 4 for background) is set to hue n2 (0 to 15) and luminance n3 (0-14, evens).
- SE.** n1 n2 n3 --
Alias for SETCOLOR.
- GR.** n --
Identical to GR. in BASIC. Adding 16 will suppress split display. Adding 32 will suppress display preclear. In addition, this GR. will not disturb player/missiles.
- POS.** x y --
Same as BASIC POSITION or POS. Positions the invisible cursor if in a split display mode, and the text cursor if in 0 GR. .
- POSIT** x y --
Positions and updates the cursor, similar to PLOT, but without changing display data.
- PLOT** x y --
Same as BASIC PLOT. PLOTS point of color in register specified by last COLOR command, at point x y.
- DRAWTO** x y --
Same as BASIC DRAWTO. Draws line from last PLOT'ed, DRAWTO'ed or POSIT'ed point to x y, using color in register specified by last COLOR command.
- DR.** x y --
Alias for DRAWTO.
- FIL** b --
Fills area between last PLOT'ed, DRAWTO'ed or POSIT'ed point to last position set by POS., using the color in register b.
- G"** --
Used in the form G" ccccc". Sends text cccc to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command prior to cccc being output. G" may be used within a colon definition, similar to ".".
- GTYPE** addr count --
Starting at addr, output count characters to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command.
- LOC.** x y -- b
Positions the cursor at x y and fetches the data from display at that position. Like BASIC LOCATE and LOC. . Note that since the word LOCATE has a different meaning in valFORTH (it is part of the advanced editor in the Utilities/Editor package), the name is not used in this package. (Advanced users: We could put Graphics in its own vocabulary, but this would add some inconvenience.)

(G") --
Run-time code compiled in by G".

POS@ -- x y
Leaves the x and y coordinates of the cursor on the stack.

CPUT b --
Outputs the data b to the current cursor position.

CGET -- b
Fetches the data b from the current cursor position.

>SCD c1 -- c2
Converts c1 from ATASCII to its display screen code, c2.
Example: ASCII A >SCD 88 @ C!
will put an "A" into the upper left corner of the display.

SCD> c1 -- c2
Converts c1 from display screen code to ATASCII c2.
See >SCD.

>BSCD addr1 addr2 count --
Moves count bytes from addr1 to addr2, translating from ATASCII
to display screen code on the way.

BSCD> addr1 addr2 count --
Moves count bytes from addr1 to addr2, translating from display
screen code to ATASCII on the way.

COLOR b --
Saves the value b in the variable COLDAT.

CLRBYT -- addr
Variable that holds data from last COLOR command.

GREY	--	0	
GOLD	--	1	
ORNG	--	2	
RDORNG	--	3	
PINK	--	4	
LVNDR	--	5	
BLPRPL	--	6	(CONSTANTS)
PRPLBL	--	7	
BLUE	--	8	
LTBLUE	--	9	
TURQ	--	10	
GRNBL	--	11	
GREEN	--	12	
YLWGRN	--	13	
ORNGRN	--	14	
LTORNG	--	15	

BOOTCOLOR hue lum --
Sets up hue for playfield 2 (text background) and lum for playfield 1
(letter intensity) in 0 Graphics mode. Lum of playfield 2 is set at 4.
After using BOOTCOLOR, doing SAVE will create a system disk with the
selected color.

Sounds

The actual production of sound by the Atari machines is rather complex and the reader is referred to the many recent (first half 1982) articles on this subject in various magazines. Here we will restrict comments to the function of the Atari audio control register. This is an eight bit register which valFORTH shadows by the variable AUDCTL. The bits have the following functions:

- bit 7: Change 17 bit polycounter to 9 bit polycounter.
Affects distortions 0 and 8.
- bit 6: Clock channel 0 with 1.79 Mhz instead of 64 Khz.
- bit 5: Clock channel 2 with 1.79 Mhz instead of 64 Khz.
- bit 4: Clock channel 1 with channel 0 instead of 64 Khz.
- bit 3: Clock channel 3 with channel 2 instead of 64 Khz.
- bit 2: Use channel 2 as crude high-pass on channel 0.
- bit 1: Use channel 3 as crude high-pass on channel 1.
- bit 0: Change normal 64 Khz to 15 Khz.

The value n may be sent to the audio control register by doing n FILTER!.

SOUND chan freq dist vol --
Sets up the sound channel "chan" as indicated.
Channel: 0-3.
Frequency: 0-255, 0 is highest pitch.
Distortion: 0-14, evens only.
Volume: 0-15.
Suggested mnemonic: CatFish Don't Vote

SO. chan freq dist vol --
Alias of SOUND.

FILTER! n --
Stores n in the audio control register and into the valFORTH shadow register, AUDCTL. Use AUDCTL when doing bit manipulation, then do FILTER!. (FILTER! does a number of housekeeping chores, so use it instead of a direct store into the hardware register.)

AUDCTL -- addr
A variable containing the last value sent to the audio control register by FILTER!. Used for bit manipulation since the audio control register is write-only.

XSND n --
Silences channel n.

XSND4 --
Silences all channels.

TEXT OUTPUT AND DISK PREPARATION GLOSSARY

- S: flag --
If flag is true, enables handler that sends text to text screen. If false, disables the handler. (See PFLAG in main glossary.) ON S: etc.
- P : flag --
If flag is true, enables handler that sends text to printer. If false, disables the handler. (See PFLAG in main glossary.) OFF P: etc.
- BEEP --
Makes a raucous noise from the keyboard. Is put in this package for lack of a better place.
- ASCII c, -- n (executing)
c, -- (compiling)
Converts next character in input stream to ATASCII code. If executing, leaves on stack. If compiling, compiles as literal.
- EJECT --
Causes a form feed on smart printers if the printer handler has been enabled by ON P:. May need adjustment for dumb or nonstandard printers.
- LISTS start count --
From start, lists count screens. May be aborted by CONSOLE button at the end of a screen.
- PLIST scr --
Lists screen scr to the printer, then restores former printer handler status.
- PLISTS start cnt --
From start, lists cnt screens to printer three to a page, then restores former printer handler status. May be aborted by CONSOLE button at the end of a screen.
- FORMAT --
With prompts, will format a disk in drive of your choice.
- (FMT) n1 -- n2
Formats disk in drive n1. Leaves 1 for good format, otherwise error number.
Note: Because of what appears to be an OS peculiarity, this operation must not be the first disk access after a boot.
- DISKCOPY1 --
With prompts, copies a source to a destination disk on single drive, with swapping. Smart routine uses all memory from PAD to bottom of Display List, producing minimum number of swaps.
- DISKCOPY2 --
With prompts, copies disk in drive 1 to disk in drive 2 using memory like DISKCOPY1.

DEBUGGING UTILITIES

DECOMP cccc

Does a decompilation of the word cccc if it can be found in the active vocabularies.

Although DECOMP is very smart, like most FORTH decompilers it will become confused by certain constructs, and will begin to print trash, with pauses in between while it looks for more trash to print. When this happens, simply hold down a CONSOLE button until DECOMP exits. This sometimes takes as much as 10 seconds, depending on luck.

CDUMP addr n --

A character dump from addr for at least n characters. (Will always do a multiple of 16.)

#DUMP addr n --

A numerical dump in the current base for at least n characters. (Will always do a multiple of 8.)

(FREE) -- n

Leaves number of bytes between bottom of display list and PAD. This is essentially the amount of free dictionary space, if additional memory is not being used for player/missiles, extra character sets, and so on.

FREE --

Does (FREE) and then prints the stack and "bytes".

H. n --

Prints n in HEX, leaves BASE unchanged.

STACK flag --

If flag is true, turns on visible stack. If flag is false, turns off visible stack.

.S ... -- ...

Does a signed, nondestructive stack printout, TOS at right. Also sets visible stack to do signed printout.

U.S ... -- ...

Does unsigned, nondestructive stack printout, TOS at right. Also sets visible stack to do unsigned printout.

B? --

Prints the current base, in decimal. Leaves BASE undisturbed.

CFALIT cccc, -- cfa (executing) cccc, -- (compiling)

Gets the cfa (code field address) of cccc. If executing, leaves it on the stack; if compiling, compiles it as a literal. Not precisely a debugging tool, but finds use in DECOMP.

FLOATING POINT WORDS

The floating-point package uses the Atari floating point routines in the operating system ROM in the same way that Atari Basic does. The routines are rather slow, and there are no trigonometric functions internal to the Atari. (SIN, COS, TAN, ATN, and ATN2 have been programmed and are available in the Advanced Graphics/Floating Point Package.) LOG and EXP are included in the operating system ROM and are supported in the present package, in base 10 and base e. Note that in the directory on screen 170 it is indicated that the ASSEMBLER must be loaded before loading the floating-point package.

Floating point words have a six byte representation in the Atari OS, and since the stack has a 60 byte maximum, a maximum of 10 floating point numbers can be on the stack at a time. In practice, this maximum often becomes 9 since some fp routines use the stack as a scratch area.

Operations involving floating-point numbers generally leave floating-point results. Exceptions are the words FIX, which takes a positive floating pointer number less than 32767.5 and leaves a rounded integer; and the floating-point comparison operators, F=, F<, etc., which leave flags. To get a floating-point number on the stack, use the word FLOATING or its alias, FP, followed by a number in Fortran "E" format. For example,

```
FP 12345
FP 12345.6
FP -12345.8
FP +5432E-16
and FP -8E18
```

will all leave floating-point numbers on the stack. Floating-point variables and constants are also supported.

It has been our experience that mistakes are common when first using this package. One must remember to use F* and not *, F+ and not +, and so on, when doing fp operations. Remember also that integers and fp numbers can't be mixed by operations: Either convert the fp number by FIX, or the integer by FLOAT, and then use the appropriate operation.

Create new words as usual. For instance, to define a floating-point square root function, write

```
: FSQRT          ( fp -- fp )
  LOG FP 2 F/ EXP ;
```

Overflow and underflow, and illegal operations such as dividing by 0, taking logarithms of negative numbers, or FIXing a negative number cause undefined and rather unpredictable results, though they do not harm the system. (Additional words in the Utilities/Editor Package cause all but one of these operations to give correct or useable results; logarithms of negatives cannot be approximated with Real numbers.)

The maximum and minimum numbers are generous, about 1E97 and 1E-97, and it is sometimes possible to exceed these limits during computation. Atari's internal representation of floating point numbers is awkward. Refer to the Atari OS manual, available from Atari, for details if needed.

FLOATING-POINT GLOSSARY

In the following, "fp" is used to indicate a floating-point number (six bytes) on the stack. The terms "top-of-stack," "2nd-on-stack" etc., have been used with the obvious meanings even though, because fp numbers are six bytes, their physical positions on the stack will not match the usual ones.

FCONSTANT cccc, fp --
 cccc: --fp

The character string is assigned the constant value fp. When cccc is executed, fp will be put on the stack.

Example: FP 3.1415926 FCONSTANT PI

FVARIABLE cccc, fp --
 cccc: addr --

The character string cccc is assigned the initial value fp. When cccc is executed, the addr (two bytes) of the value of cccc will be put on the stack.

Example: FP 0 FVARIABLE X
 FP 18.4 X F!

FDUP fp1 -- fp1 fp1
Copies the fp number at top-of-stack.

FDROP fp --
Discards the fp number at top-of-stack.

FOVER fp2 fp1 -- fp2 fp1 fp2
Copies the fp number at 2nd-on-stack to top-of-stack.

FLOATING cccc, -- fp
Attempts to convert the following string, cccc, to a fp number. Stops on reaching first unconvertible character and skips the rest of the string. If no characters convertible, leaves unpredictable fp number on stack.

FP cccc, --fp
Alias for FLOATING.

F@ addr -- fp
Fetches the fp number whose address is at top-of-stack.

F! fp addr --
Stores fp into addr. Remember that the operation will take six bytes in memory.

F. fp --
Type out the fp number at top-of-stack. Ignores the current value in BASE and uses base 10.

F? addr --
Fetches a fp number from addr and types it out.

F+ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their fp sum, fp3.

F- fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their difference, fp3=fp2-fp1.

F* fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their product, fp3.

F/ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their quotient, fp3=fp2/fp1.

FLOAT n -- fp
Replaces number at top-of-stack with its fp equivalent.

FIX fp (non-neg, less than 32767.5) -- n
Replaces fp number at top-of-stack, constrained as indicated, with its integer equivalent.

LOG fp1 -- fp2
Replaces fp1 with its base e logarithm, fp2. Not defined for fp1 negative.

LOG10 fp1 -- fp2
Replaces fp1 with its base 10 decimal logarithm, fp2. Not defined for fp1 negative.

EXP fp1 -- fp2
Replaces fp1 with fp2, which equals e to the power fp1.

EXP10 fp1--fp2
Replaces fp1 with fp2, which equals 10 to the power fp1.

F0= fp -- flag
If fp is equal to floating-point 0, a true flag is left. Otherwise, a false flag is left.

F= fp2 fp1 -- flag
If fp2 is equal to fp1, a true flag is left. Otherwise, a false flag is left.

F> fp2 fp1 -- flag
If fp2 is greater than fp1, a true flag is left. Otherwise, a false flag is left.

F< fp2 fp1 -- flag
If fp2 is less than fp1, a true flag is left. Otherwise, a false flag is left.

FLITERAL fp --
If compiling, then compile the fp stack value as a fp literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

 : xxx [calculate] FLITERAL ;
Compilation is suspended for the compile time calculation of a value.
Compilation is resumed and FLITERAL compiles the value on stack.

FLIT -- fp

Within a colon definition, FLIT is automatically compiled before each fp number encountered as input text. Later execution by the system of FLIT as it is encountered in the dictionary cause the context of the next 6 dictionary addresses to be pushed to the stack as a fp number. FLIT is also compiled in explicitly by FLITERAL.

ASCF addr -- fp

An ASCII-to-floating-point conversion routine. Uses Atari OS routine. The routine reads string starting at addr and attempts to create a floating point number. If string is not a valid ASCII floating-point representation, leaves undefined result on stack. Used by FLOATING.

FS fp --

System routine. Sends fp argument on stack to Atari register FRO. Experts only.

>F -- fp

System routine. Fetches fp argument from Atari register FRO. Experts only.

<F fp1 fp2 --

System routine. Sends fp1 and fp2 to Atari registers FR1 and FRO respectively. Experts only.

F.TY --

System routine. Types out last fp number converted by FASC.

CIX addr --

System variable. One byte offset pointer in buffer pointed to by INBUF. Experts only.

INBUF addr --

System variable. Used by ASCF to know where ASCII string to be converted is located.

FR1 -- n

System constant. Atari internal register address.

FRO --n

System constant. Atari internal register address.

FPOLY addr count --

A system routine for advanced users doing polynomial evaluation.

The polynomial $P(Z) = \text{SUM}(i=0 \text{ to } n) (A(i)*Z^{**i})$ is computed by the following standard method:

$$P(Z) = (... (A(n)*Z + A(n-1))*Z + ... + A(1))*Z + A(0)$$

The address addr points to the coefficients A(i) stored sequentially in memory, with the highest order coefficient first. The count is the number of coefficients in the list. The independent variable Z, in floating-point, should be sent to FRO using FS. FPOLY is then executed. The result put on the stack using >F. Note that FPOLY is intended to be used in a Forth word.

Trigonometric functions and general polynomial expansions, for example, may be defined more simply with the help of this routine.

FLG10 --
System routine used by LOG10.

FLG --
System routine used by LOG.

FEX --
System routine used by EXP.

FEX10 --
System routine used by EXP10.

FDIV --
System routine used by F/.

FMUL --
System routine used by F*.

FSUB --
System routine used by F-.

FADD --
System routine used by F+.

FPI --
System routine used by FIX.

IFP --
System routine used by FLOAT.

FASC --
System routine, Does floating-point-to-ASCII conversion on the fp number in FRO and leaves string at address pointed to by INBUF. Last byte of string has most significant bit set. Used by F.TY.

AFP --
System routine used by ASCF.

(intentionally left blank)

OPERATING SYSTEM

This package implements the computer's Operating System I/O routines. The 850 (RS-232C) driver package may be loaded into the dictionary by using the word RS232, which will then support references to devices "R1" through "R4."

The code for this section was originally written by Patrick Mullarky, and published through the Atari Program Exchange. It is used here by permission of the author.

OS GLOSSARY

OPEN addr n1 n2 n3 -- n4

This word opens the device whose name is at addr. The device is opened on channel n3 with AUX1 and AUX2 as n1 and n2 respectively. The device status byte is returned as n4. The name of a device may be produced in various ways: For a single character name, say "S" for the screen handler,

 ASCII S PAD C!

will leave the ASCII value of S at PAD. Then

 PAD 8 0 3 OPEN

will open the screen handler on channel 3 with AUX1 = 8 (write only) and AUX2 = 0. If you have the UTILITIES/EDITOR Package, longer names may be set up simply by using the word " " .

CLOSE n --

Closes channel n.

PUT b1 n -- b2

Outputs byte b1 on channel n, returns status byte b2.

GET n -- b1 b2

Gets byte b1 from channel n, returns status byte b2.

GETREC addr n1 n2 -- n3

Inputs record from channel n2 up to length n1. Returns status byte n3.

PUTREC addr n1 n2 -- n3

Outputs n1 characters starting at addr through channel n2. Returns status byte n3.

STATUS n -- b

Returns status byte b from channel n.

DEVSTAT n -- b1 b2 b3

From channel n1 gets device status bytes b1 and b2, and normal status byte b3.

SPECIAL b1 b2 b3 b4 b5 b6 b7 b8 -- b9

Implements the Operating System "Special" command. AUX1 through AUX6 are b1 through b6 respectively, command byte is b7, channel number is b8. Returns status byte b9.

RS232 --

Loads the Atari 850 drivers into the dictionary (approx 1.8K) through a three-step bootstrap process. Executing this command more than once without turning the 850 off and on again will crash the system.

valForth Glossary

Based on the fig-Forth Glossary
Provided through the courtesy of
Fourth Interest Group, P.O. Box 1105, San Carlos, CA 94070

This glossary contains all of the word definitions in Release 1.1 of valForth. The definitions are presented in the order of their ASCII sort.

The first line of each entry shows a symbolic description of the action of the procedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Two dashes "--" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. hi 8 bits zero)
c	7 bit ASCII character (hi 9 bits zero)
d	32 bit signed double integer, most significant portion with sign on top of stack.
f	boolean flag. 0=false, non-zero=true
tf	boolean true flag=non-zero
ff	boolean false flag=0
n	16 bit signed integer number
u	16 bit unsigned integer

The capital letters on the right show definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.
E	Intended for execution only.
L0	Level Zero definition of FORTH-78
L1	Level One definition of FORTH-78
P	Has precedence bit set. Will execute even when compiling. (immediate)
U	A user variable.
V	A valForth word not in fig-Forth.
B	A word adopted from Leo Brodie's <u>Starting Forth</u> .

Unless otherwise noted, all references to numbers are for 16 bit signed integers. The high byte of 16 bit numbers is the second byte on the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16 bit signed integer math, with error and underflow indication unspecified.

! n addr -- L0
Store 16 bits of n at address. Pronounced "store".

!CSP --
Save the stack position in CSP. Used as part of the compiler security.

d1 -- d2 L0
Generate from a double number d1, the next ASCII character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S. Pronounced "number".

#> d -- addr count L0
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE. Pronounced "number-bracket".

#S d1 -- d2 L0
Generates ASCII text in the text output buffer, by the use of #, until a zero double number d2 results. Used between <# and #>. Pronounced "numbers".

' -- addr P,L0
Used in the form:
' nnnn
Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

'(-- V,E,P
Used in the form:
'(WORD0 WORD1 . . . WORDN)(WORDN+1 . . . WORDM)
which executes as follows: If WORD0 is found in a search of CONTEXT and CURRENT, then execute WORD1 . . . WORDM. Generally used for conditional compilation. Note that if no words are to be included in the second group, then)(and) must be separated by at least TWO blanks.

(-- P,L0
Used in the form:
(cccc)
Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

) -- V,E,P
No operation. Used in '(constructs.

)() -- V,E,P
Scans text input pointer past ")". Used in '(constructs.

(. ") -- C+
The run-time procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

```

(/LOOP)          n --                                B,C2
                  Execution time code of /LOOP.

( ;CODE)          --                                C
                  The run-time procedure, compiled by ;CODE, that rewrites the code
                  field of the most recently defined word to point to the following
                  machine code sequence. See ;CODE.

(+LOOP)          n --                                C2
                  The run-time procedure compiled by +LOOP, which increments the loop
                  index by n and tests for loop completion. See +LOOP.

(ABORT)          --
                  Executes after an error when WARNING is -1. This word normally exe-
                  cutes ABORT, but may be altered (with care) to a user's alternative
                  procedure.

(DO)             --                                C
                  The run-time procedure compiled by DO which moves the loop control
                  parameters to the return stack. See DO.

(FIND)           addr1 addr2 -- pfa b tf             (ok)
                  addr1 addr2 -- ff                 (bad)
                  Searches the dictionary starting at the name field address addr2,
                  matching to the text at addr1. Returns parameter field address,
                  length byte of name field and boolean true for a good match. If no
                  match is found, only a boolean false is left.

(FMT)            n1 -- n2                            V
                  Formats disk in drive n1. Leaves 1 for good format, otherwise error
                  number. Note: Because of what appears to be an OS peculiarity, this
                  operation must not be the first disk access after a boot.

(LINE)           n1 n2 -- addr count
                  Convert the line number n1 and the screen n2 to the disc buffer address
                  containing the data.

(LOOP)           --                                C2
                  The run-time procedure compiled by LOOP which increments the loop
                  index and tests for loop completion. See LOOP.

(NUMBER)         d1 addr1 -- d2 addr2
                  Convert the ASCII text beginning at addr1+1 with regard to BASE.
                  The new value is accumulated into double number d1, being left as d2.
                  Addr2 is the address of the first unconvertible digit. Used by NUMBER.

(SAVE)           --                                V
                  Used by SAVE, not generally used in programs. Sets up various
                  parameters preparatory to writing a bootable disk.

*               n1 n2 -- prod                        L0
                  Leave the signed product of two signed numbers. Pronounced "star".

*/             n1 n2 n3 -- n4                        L0
                  Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention
                  of an intermediate 31 bit product permits greater accuracy than would
                  be available with the sequence:
                  n1 n2 * n3 /
                  Pronounced "star-slash".

```

***/MOD** n1 n2 n3 -- n4 n5 L0
 Leave the quotient n5 and remainder n4 of the operation n1*n2/n3.
 A 31 bit intermediate product is used as for */. Pronounced "star slash-mod".

+ n1 n2 -- sum L0
 Leave the sum n1+n2.

+! n addr -- L0
 Add n to the value at the address. Pronounced "plus-store".

+ - n1 n2 -- n3
 Apply the sign of n2 to n1, which is left as n3.

+BUF addr1 -- addr2 f
 Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP n1 -- (run)
 addr n2 -- (compile) P,C2,L0
 Used in a colon-definition in the form:
 DO . . . n1 +LOOP
 At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n -- addr
 Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, n -- L0
 Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

- n1 n2 -- diff L0
 Leave the difference of n1-n2.

--> -- P,L0
 Continue interpretation with the next disc screen. Pronounced "next-screen".

-DISK addr n2 n3 flag -- n4 V
 Used by R/W. Not generally used in programs. This word performs a single-sector read or write on a disk. Addr is the starting RAM address, n2 is the sector number (1-720), n3 is the drive number (1-4), and the flag is 1 for read and 0 for write. On return, n4 will be zero if there were no problems, or it will be a DOS error number if a DOS error occurred.

-DUP

LO

-FIND

Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true are left. Otherwise, only a boolean false is left.

$$\text{addr } n1 \quad \text{--} \quad \text{addr } n2$$

n ---

Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blanks follows. Pronounced "dot".

P.L0

Compiles an in-line string cccc (delimited by the trailing ") with an execution procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". The maximum number of characters may be an installation dependent value. See (.").

```
line scr --
```

.R

Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

```
n1  n2  --  quot
```

LO

Leave the signed quotient of $n1/n2$.

n = 1000

B.C2

Like +LOOP, but uses an unsigned limit, index, and increment. Thus loop index may pass \$7FFF without mishap. Faster than +LOOP and may be used instead of +LOOP if increment is positive (and index doesn't cross \$7FFF.)

```
n1  n2  --  rem  quot
```

LO

Leave the remainder and signed quotient of $n1/n2$. The remainder has the sign of the dividend.

0 1 2 3

-- n

These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0#

n -- flag

V

Leaves a true flag if n is not equal to 0. Otherwise, leaves a false flag. Pronounced "zero not equal".

0<

n -- f

L0

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0=

n -- f

L0

Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0>

n -- flag

V

Leaves a true flag if n is greater than 0. Otherwise leaves a false flag.

OBRANCH

f --

C2

The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+

n1 -- n2

L1

Increment n1 by 1.

1-

n1 -- n2

B

Subtract one from n1. Pronounced "one-minus".

2*

n1 -- n2

B

Multiply n1 by two. Pronounced "two-star" or "two-times".

2+

n1 -- n2

Leave n1 incremented by 2.

2-

n1 -- n2

Leave n1 decremented by 2.

2/

n1 -- n2

B

Divide n1 by two. Pronounced "two-slash".

2DROP

d --

B

Drops the double number at TOS.

2DUP

d -- d d

B

Copies double number at TOS.

2OVER

d2 d1 -- d2 d1 d2

B

Copies double number at 2OS to TOS.

2ROT

d3 d2 d1 -- d2 d1 d3

V

Moves double number at 3OS over two double numbers on 2OS and TOS.

2SWAP

d2 d1 -- d1 d2

B

Exchanges double numbers at TOS and 2OS.

:	--	P,E,L0
Used in the form called a colon-definition:		
	: cccc ... ;	
Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '...' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.		
;	--	P,C,L0
Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.		
;CODE	--	P,C,L0
Used in the form:		
	: cccc ;CODE assembly mnemonics	
Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to Assembler, assembling to machine code the following mnemonics.		
When cccc later executes in the form:		
	cccc nnnn	
the word nnnn will be created with its execution procedure given by the machine code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.		
;S	--	P,L0
Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.		
<	n1 n2 -- f	L0
Leave a true flag if n1 is less than n2; otherwise leave a false flag.		
<#	--	L0
Set-up for pictured numeric output formatting using the words:		
	<# # #S SIGN #>	
The conversion is done on a double number producing text at PAD. Pronounced "Bracket Number".		
<=	n2 n1 -- flag	V
Leaves true flag if n2 is less than or equal to n1. Otherwise, leaves false flag.		
<>	n2 n1 -- flag	V
Leaves true flag if n2 and n1 are unequal. Otherwise, leaves false flag.		

<BUILDS

--

C,L0

Used within a colon-definition:

```
: cccc <BUILDS ...  
DOES> ... ;
```

Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. Executing cccc in the form:

```
cccc nnnn
```

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow run-time procedures to be written in high-level rather than in assembler code (as required by ;CODE).

=

```
n1 n2 -- f
```

L0

Leave a true flag if n1=n2; otherwise leave a false flag.

=>

--

If using 512 byte (half-K) screen, does --> otherwise, no action is taken. Used to chain screens in half-K format that will still load correctly in valFORTH full-K format.

>

```
n1 n2 -- f
```

L0

Leave a true flag if n1 is greater than n2; otherwise a false flag.

>=

```
n2 n1 -- flag
```

V

Leaves true flag if n2 is greater than or equal to n1. Otherwise, leaves false flag.

>R

```
n --
```

C,L0

Remove a number from the computation stack and place as the most accessible on the return stack. Use should be balanced with R> in the same definition.

?

```
addr --
```

L0

Print the value contained at the address in free format according to the current base.

?1K

```
-- flag
```

V

Leaves a true flag if C/L is 64, indicating 1K screens. Otherwise, leaves a false flag.

?COMP

--

Issue error message if not compiling.

?CSP

--

Issue error message if stack position differs from value saved in CSP.

?ERROR

```
f n --
```

Issue an error message number n, if the boolean flag is true.

?EXEC

--

Issue an error message if not executing.

?EXIT

--

V,C

Caution: Use only within a DO LOOP. Within DO LOOP, will cause exit at end of current loop if a CONSOLE button is depressed when ?EXIT is

executed. Will work only in the word in which the DO LOOP is defined, not in a word nested further down.

?LOADING

--
Issue an error message if not loading.

?PAIRS

n1 n2 --
Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

--
Issue an error message if the stack is out of bounds. This definition may be installation dependent.

?TERMINAL

-- b
Perform a test of the terminal keyboard for actuation of a CONSOLE key. Leaves 0 if none actuated, leaves 1 for START, 2 for SELECT, 4 for OPTION, and sums for combinations.

@

addr -- n L0
Leave the 16 bit contents of address.

@EX

addr -- V
Fetches the word (presumably a code field address) at addr, and then causes it to execute. Used for conditional execution without the speed and memory loss of flags and/or case statements. Typical use would be AUXOP @EX where the variable AUXOP had been loaded with the cfa of the desired word, by ' DESIREDWORD CFA AUXOP ! . Pronounced "fetch-ex."

ABORT

-- L0
Clear the stacks and enter the execution state. Return control to the operators terminal, printing a message appropriate to the installation.

ABS

n -- u L0
Leave the absolute value of n as u.

ACCEPT

addr count --
Same as EXPECT, except that ACCEPT uses the O.S. line input routine which allows full MEMO PAD editing functions to be utilized. EXPECT prevents hazards such as Shift-Clear while ACCEPT does not. SEE EXPECT.

AGAIN

addr n -- (compiling) P,C2,L0
Used in a colon-definition in the form:
BEGIN ... AGAIN
At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

BOOT	--	V
Boots disk in drive 1. Same effect as turning computer on and off.		
BRANCH	--	C2,L0
The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.		
BUFFER	n -- addr	
Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc. The block is not read from the disc. The address left is the first byte within the buffer for data storage.		
C!	b addr --	
Store 8 bits at address.		
C,	b --	
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.		
C/L	-- n	V
A CONSTANT equal to the number of characters per line in the ValForth screen. Usually 32, but may be 64 if 1024 byte screens in use.		
C?	addr --	V
Fetches a byte from addr and prints it using . .		
C@	addr --- b	
Leave the 8 bit contents of memory address.		
CFA	pfa --- cfa	
Convert the parameter field address of a definition to its code field address.		
CMOVE	from to count ---	
Move the specified quantity of bytes beginning at address from to address to. The contents of address from is moved first proceeding toward high memory.		
COLD	--	
The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.		
COMPILE	--	C2
When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).		
CONSTANT	n --	L0
A defining word used in the form: n CONSTANT cccc to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.		

CONTEXT	-- addr	U,L0
	A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.	
COUNT	addr1 -- addr2 n	L0
	Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically, COUNT is followed by TYPE.	
CR	--	L0
	Transmit a carriage return and line feed to the selected output device.	
CREATE	--	
	A defining word used in the form: CREATE cccc by such words as CODE and CONSTANT to create a dictionary header for a FORTH definition. The code field contains the address of the word's parameter field. The new word is created in the CURRENT vocabulary.	
CSAVE	--	V
	Creates a bootable copy of RAM-resident system up to HERE on cassette. Computer beeps twice to indicate user must press Record and Play buttons on recorder, prior to pressing RETURN. CSAVE expects leaderless tape. If your tape has a leader, wind to just before the end of leader.	
CSP	-- addr	U
	A user variable temporarily storing the stack pointer position, for compilation error checking.	
CURRENT	-- addr	
	Address of a pointer to second word in the parameter field of the current vocabulary. (The current vocabulary is the one to which new definitions are added.)	
D!	d addr --	V
	Stores double number d into addr.	
D+	d1 d2 -- dsum	
	Leave the double number sum of two double numbers.	
D+-	d1 n -- d2	
	Apply the sign of n to the double number d1, leaving it as d2.	
D.	d --	L1
	Print a signed double number from a 32 bit two's complement value. The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced "D-dot."	
D.R	d n --	
	Print a signed double number d right aligned in a field n characters wide.	
D@	addr -- d	V
	Fetches double number d from addr.	


```
DPL      -- addr                                     U,L0
A user variable containing the number of digits to the right of the
decimal on double integer input. It may also be used hold output column
location of a decimal point, in user generated formatting. The default
value on single number input is -1.
```

Installation dependent commands to select disc drives, by presetting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection. Offset is suppressed for error text so that it may always originate from drive 0.

DROP	n --	LO
	Drop the number from the stack.	

DUP	n -- n n	LO
	Duplicate the value on the stack.	

ELSE addr1 n1 -- addr2 n2 (compiling) P,C2,L0

Occurs within a colon-definition in the form:

IF ... ELSE ... ENDIF

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time, ELSE emplaces BRANCH reserving a branch offset, leaves the address `addr2` and `n2` for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from `addr1` to HERE and storing at `addr1`.

```

EMIT          c  --                                LO
Transmit ASCII character c to the selected output device.  OUT is
incremented for each character output.

```

EMPTY-BUFFERS	L0
Mark all block-buffers as empty, not necessarily affecting the contents. Up-dated blocks are not written to the disc. This is also an initiali- zation procedure before first use of the disc. Alias is MTB.	

ENCLOSE addr1 c -- addr1 n1 n2 n3

The text scanning primitive used by WORD. From the text address addr1 and an ASCII delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ASCII "null", treating it as an unconditional delimiter.

END P,C2,L0

This is an "alias" or duplicate definition for UNTIL.

ENDIF addr n -- (compile) P,CO,LO
 Occurs in a colon-definition in form:
 IF ... ENDIF
 IF ... ELSE ... ENDIF
 At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also, IF and ELSE.
 At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ERASE addr n --
 Clear a region of memory to zero from addr over n addresses.

ERROR n -- in blk
 Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 176 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 176. If WARNING=0, n is just printed as a message number (non disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

EXECUTE addr --
 Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count -- LO
 Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE -- addr U
 A user variable containing an address below which FORGETting is trapped. To forget below this point, the user must alter the contents of FENCE.

FILL addr quan b --
 Fill memory at the address with the specified quantity of bytes b.

FIRST -- addr
 A constant that leaves the address of the first (lowest) block buffer.

FLD -- addr U
 A user variable for control of number output field width. Presently unused in fig-FORTH.

FORGET -- E,LO
 Executed in the form:
 FORGET cccc
 Delete definition named cccc from the dictionary with all entries physically following it.

FORTH	--	P,L1
	The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.	
FULLK	--	V
	Sets C/L to 64 and B/SCR to 8, producing 1024 byte screen operation. May be SAVED in this condition. (See HALFK)	
GFLAG	-- addr	V
	A variable that holds a Graphics mode cursor control flag. When the value at GFLAG is non-zero, valForth assumes a split-screen is operative, and will use the alternate cursor-address variables provided by the Operating System to use the text window at the bottom of the display.	
HALFK	--	V
	Sets C/L to 32 and B/SCR to 4, producing 512 byte screen operation. May be SAVED in this condition. (See FULLK)	
HERE	-- addr	LO
	Leave the address of the next available dictionary location.	
HEX	--	LO
	Set the numeric conversion base to sixteen (hexadecimal).	
HLD	-- addr	LO
	A user variable that holds the address of the latest character of text during numeric output conversion.	
HOLD	c --	LO
	Used between <# and #> to insert an ASCII character into a pictured numeric output string. e.g. 2E HOLD will place a decimal point.	
I	-- n	C,LO
	Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent. See R.	
I'	-- n	B
	Copies the second item on the return stack to the stack. Generally used to get the index of the present DO LOOP after an item has been pushed to the return stack for convenience.	
ID.	addr --	
	Print a definition's name from its name field address.	

IF f -- (run-time)
 -- addr n (compile) P,C2,L0
 Occurs in a colon-definition in form:
 IF (tp) ... ENDIF
 IF (tp) ... ELSE (fp) ... ENDIF
 At run-time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional.; if missing, false execution skips to just after ENDIF.
 At compile-time IF compiles OBRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE --
 Mark the most resently made definition so that when encountered at compile time, it will be executed rather than being compiled, i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceeding it with [COMPILE].

IN -- addr L0
 A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX from to --
 Print the first line of each screen over the range from, to. This is used to view the comment (first) lines of an area of text on disc screens.

INTERPRET --
 The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with a "?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER.

J -- n B
 Copies the third item on the return stack to the stack. Generally used to get the index of the next outer DO LOOP.

KEY -- c L0
 Leave the ASCII value of the next terminal key struck.

KLOAD screen -- V
 If C/L has a value other than 64, then the number on stack is doubled. In either case, LOAD is then executed. The purpose is to allow smart conditional loading of either 1K screen or 1/2K screen formats. See '(.

LABEL

cccc, --
cccc, -- addr

At compilation time, creates a word cccc. At run time, cccc leaves the address of its pfa on the stack. Used to set up a pointer to the following area of memory, as for a machine language subroutine or a player image. Examples:

Example 1: Player image

```
2 BASE !  
LABEL UPARROW  
00011000 C,  
00111100 C,  
01111110 C,  
00011000 C,  
00011000 C,  
00011000 C,  
DCX
```

Example 2: Machine code two-times

```
ASSEMBLER  
LABEL 2* 0 ,X ASL, 1 ,X ROL, RTS,
```

LATEST

-- addr

Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE

--

C,L0

Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA

pfa -- lfa

Convert the parameter field address of a dictionary definition to its link field address.

MESSAGE

Print on the selected output device the text of line n relative to screen 176 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc unavailable).

MIN

LO

Leave the smaller of two numbers.

MINUS

LO

Leave the two's complement of a number.

MOD

LO

Leave the remainder of $n1/n2$, with the same sign as $n1$.

MTB

V

Alias of EMPTY-BUFFERS.

NEXT

This is the inner interpreter that uses the interpretive pointer IP to execute compiled Forth definitions. It is not directly executed, but is the return point for all code procedures. It acts by fetching the address pointed to by IP, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth. (Assembler Vocabulary)

NFA

Convert the parameter field address of a definition to its name field.

NOOP

V

A word that does nothing in minimal time. May be used for reserving space in a definition or as a null operation for a word that uses @EX. Generally for advanced programmers. Identical to TASK. Pronounced "no-op".

NOT

V

Leaves a true flag if n is equal to 0. Otherwise, leaves a false flag.

NUMBER

Convert a character string left at addr with a preceeding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

0+S

V

Same as OVER + SWAP. Used to set up limits for DO LOOPS and the like.

OFF

V

A CONSTANT equal to 0. Used to enhance readability.

OFFSET	-- addr	U
	A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DRO, DR1, MESSAGE.	
ON	-- 1	V
	A CONSTANT equal to 1. Used to enhance readability.	
OR	n1 n2 -- or	LO
	Leave the bit-wise logical or of two 16 bit values.	
OUT	-- addr	U
	A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.	
OVER	n1 n2 -- n1 n2 n1	LO
	Copy the second stack value, placing it as the new top.	
PAD	-- addr	LO
	Leave the address of the text output buffer, which is a fixed offset above HERE.	
PFA	nfa -- pfa	
	Convert the name field address of a compiled definition to its parameter field address.	
PFLAG	-- addr	V
	A variable that holds an output-select value. If bit 0 is set then output will be sent to the display screen. If bit 1 is set, then output will be sent to the printer. If both bits are set, then output will go to both channels.	
PICK	... n -- ... n1	V
	Copies the nth entry below n on stack to top of stack. 2 PICK is the same as OVER, 1 PICK is the same as DUP.	
POP	-- addr	
	The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is a Forth re-entry point after machine code. (Assembler Vocabulary)	
PREV	-- addr	
	A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.	
PROMPT	--	V
	Intended for system use only, in QUIT. A smart version of the usual ." ok" in QUIT. Prevents "ok" and visible stack printout from being routed to printer.	
PUSH	-- addr	
	This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code. (Assembler Vocabulary)	

```

PUT          -- addr
This code sequence stores machine register contents over the topmost
computation stack value and returns to NEXT. It is not directly execu-
table, but is a Forth re-entry point after machine code.

QUERY       --
Input 80 characters of text (or until a "return") from the operators
terminal. Text is positioned at the address contained in TIB with IN
set to zero.

QUIT        --                                L1
Clear the return stack, stop compilation, and return control to the
operators terminal. No message is given.

R           -- n
Copy the top of the return stack to the computation stack.

R#          -- addr                                U
A user variable which may contain the location of an editing cursor,
or other file related function.

R/W         addr blk f --
The fig-FORTH standard disc read-write linkage. addr specified is the
source or destination block buffer, blk is the sequential number of
the referenced block; and f is a flag for f=0 write and f=1 read.
R/W determines the location on mass storage, performs the read-write
and performs any error checking. Important: See Note 1 at end of glossary.

R>          -- n                                L0
Remove the top value from the return stack and leave it on the compu-
tation stack. See >R and R.

R0          -- addr                                U
A user variable containing the initial location of the return stack.
Pronounced R-zero. See RP!

REPEAT      addr n -- (compiling)                P,C2
Used within a colon-definition in the form:
  BEGIN ... WHILE ... REPEAT
At run-time, REPEAT forces an unconditional branch back to just after
the corresponding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to
addr. n is used for error testing.

ROLL        ... n -- ...                        V
Moves the nth entry below n on stack to top of stack. 3 ROLL is the
same as ROT, 2 ROLL is the same as SWAP. 0 ROLL is undefined.

ROT         n1 n2 n3 --- n2 n3 n1                L0
Rotate the top three values on the stack, bringing the third to the
top.

RP!         --
A computer dependent procedure to initialize the return stack pointer
from user variable R0.

```


TYPE

addr count -- L0
Transmit count characters from addr to the selected output device.

TYPE as supplied zeroes out the high bit of each character before sending it to the output device, usually the screen or printer. If you want to be able to type all 8 bits for inverse characters, do the following:

255 ' TYPE 20 + C!

and return to 7 bit output by doing

127 ' TYPE 20 + C!

More generally,

```
: 78TYPE < ' TYPE 14 + > C! ;  
: 7TYPE 127 78TYPE ;  
: 8TYPE 255 78TYPE ;
```

(What you are doing with all of this is changing the mask that TYPE uses before executing EMIT.)

U* u1 u2 -- ud
Leave the unsigned double number product of two unsigned numbers.

U. n -- B
Prints the number n in unsigned form.

U.R u n -- B
Prints unsigned number u right justified in a field n wide.

U/ ud u1 -- u2 u3
Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

U> u2 u1 -- flag V
Leaves true flag if u2 (unsigned) is greater than u1 (unsigned). Otherwise, leaves false flag.

VLIST --
List the names of the definitions in the context vocabulary.

WAIT -- V
Halts execution of Forth until a CONSOLE button is pressed.

WARNING -- addr U
A user variable containing a value controlling messages. If = 1 disc is present, and screen 176 of drive 0 is the base location for messages. If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified procedure. See MESSAGE, ERROR.

WHILE f -- (run-time) P,C2
 ad1 n1 -- ad1 n1 ad2 n2
Occurs in a colon-definition in the form:
 BEGIN ... WHILE (tp) ... REPEAT
At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE emplaces (OBRANCH) and leaves ad2 of the reserved offset. The stack value will be resolved by REPEAT.

WIDTH -- addr U
In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions' name. It must be 1 thru 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD c -- L0
Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurrences of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X --
This is pseudonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the execution procedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

XOR n1 n2 -- xor L1
Leave the bit-wise logical exclusive-or of two values.

ok -- V
Does three backspaces. When using "logical line input" from keyboard to re-input previously entered line, this word gives harmless meaning to the old "ok" prompt Forth may find in the input stream. See "logical line input" section in Strolling through ValForth.

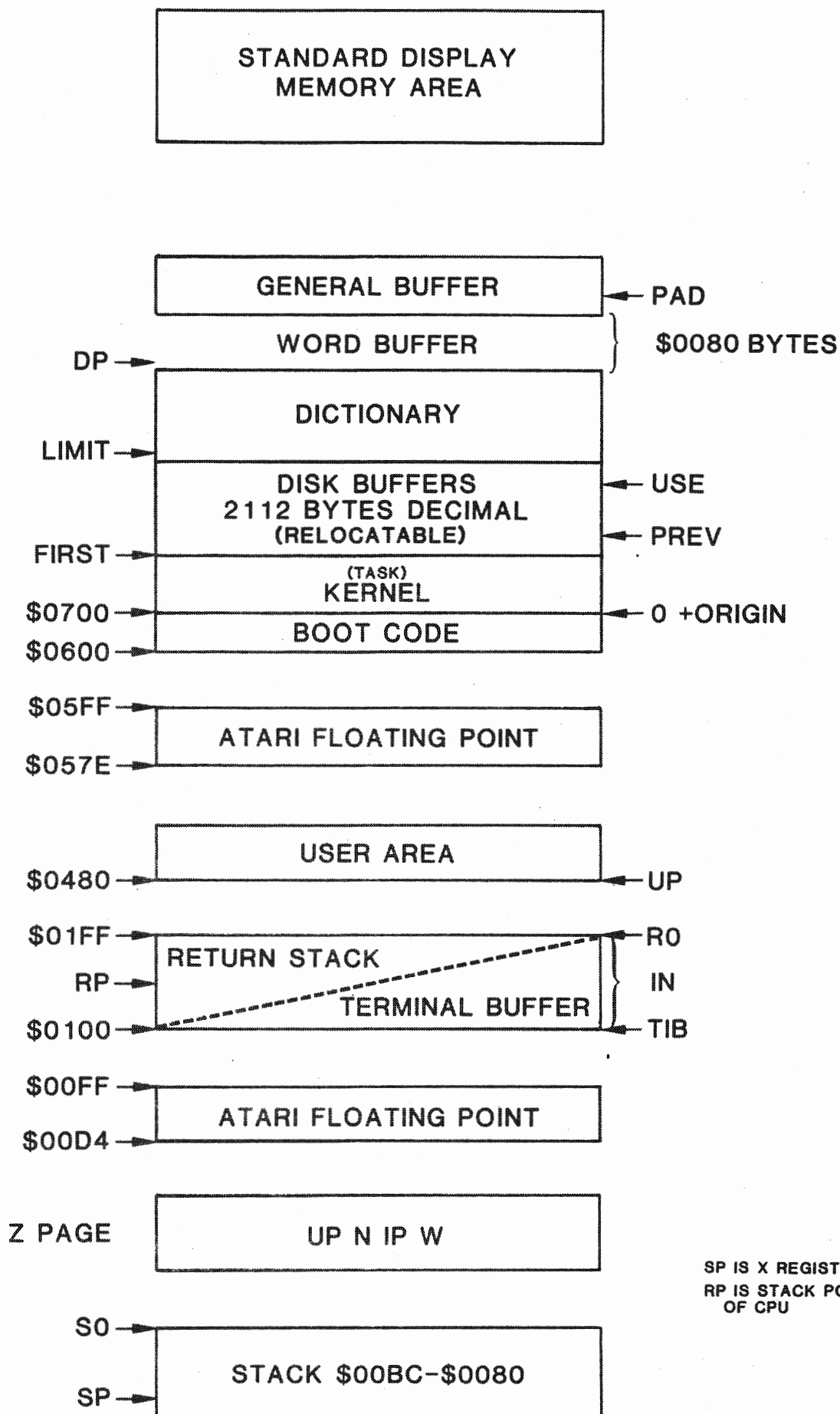
[--	P,L1
	Used in a colon-definition in form:	
	: xxx [words] more ;	
	Suspend compilation. The words after [are executed, not compiled.	
	This allows calculation or compilation exceptions before resuming	
	compilation with]. See LITERAL,].	
[Compile]	--	P,C
	Used in a colon-definition in form:	
	: xxx [COMPILE] FORTH ;	
	[COMPILE] will force the compilation of an immediate definition, that	
	would otherwise execute during compilation. The above example will	
	select the FORTH vocabulary when xxx executes, rather than at compile	
	time.	
]	--	L1
	Resume compilation, to the completion of a colon-definition. See [.	

NOTES:

Note 1

Due to a bug in at least some of the Atari Operating System ROM programs, a sector may not be written directly from a memory area in which the low byte of the bottom location is \$7F. The system will hang if this is attempted. This is not a valFORTH bug, it is Atari's. Please watch out for it.

valFORTH^{T.M.} Memory Map



vaIFORTH

T.M.

Advanced 6502 Macro Assembler

Version 2.0
April 1982

Although the FORTH language is many times faster than BASIC or PASCAL, there are still times when speed is so critical that one must turn to assembly language programming as a matter of necessity. Not wanting to give up the advantages of the FORTH language, FORTH programmers typically use an assembler designed specifically for the FORTH system. vaIFORTH incorporates a very powerful FORTH style 6502 assembler for these special programming jobs.

Overview

Most programming applications can be undertaken completely in high level FORTH. There are times, due to speed constraints, when assembly language must be used. Typically, "number crunching" and high speed graphic routines must be machine coded. valFORTH provides a powerful 6502 FORTH assembler for these special occasions.

FORTH assemblers differ from standard assemblers by making the best use of the stack and the FORTH system as a whole. The FORTH assembler is smaller than a standard assembler. In the case of the valFORTH assembler, this is particularly true.

The valFORTH assembler offers the programmer the following improvements over a standard assembler:

- 1) IF...THEN...ELSE structures which use positive logic rather than negative logic.
- 2) BEGIN...UNTIL structures for post-testing indefinite loops.
- 3) WHILE...REPEAT structures for pre-testing indefinite loops.
- 4) BEGIN...AGAIN structures for unconditional looping.
- 5) Full access to the FORTH operating system and its capabilities such as changing bases.
- 6) Complex assembly time calculations.
- 7) Mixed high level FORTH with assembly code to take full advantage of each.
- 8) Full macro capability.

The following is a complete description of the valFORTH assembler. This description assumes a working knowledge of 6502 assembly language programming and related terms.

The purpose of the FORTH assembler is to allow machine language programming without the need to abandon the FORTH system. Words coded in assembly language must follow the standard FORTH dictionary format and must adhere to certain guidelines regarding their coding.

Assembly language programmers typically have two methods of storing programs into RAM. The machine code can be poked directly into memory, or an assembler can be used to accomplish this. The former method is brutal, but it has the advantage that precious memory is not taken up by the assembler. The drawback, of course, is loss of readability and ease of modification. FORTH allows both of these methods to be employed.

The words ":", "C," and "C;" can be used to poke any machine language program into the dictionary. This is used only when memory restrictions prohibit the use of an assembler or if it is assumed that no assembler is available.

In high level FORTH, words are compiled into the dictionary using the following form:

```
: name      high-level-FORTH... ;
```

When compiling a machine coded word, this becomes:

```
CODE name      machine-code... C;
```

In this example, the word "CODE" creates a header for the next word in much the same way ":" creates a header. The difference lies in the fact that ":" informs the system that the following definition is high level FORTH, while "CODE" indicates that the definition is a machine or assembly language definition. In the same manner, ";" terminates a high level FORTH definition while "C;" terminates a code definition.

To clarify this, a code definition will be programmed that will clear the top line of the current video display on an Atari 800 microcomputer. Note that video memory is pointed to by the address stored in locations 88 and 89 (decimal). The 6502 code is shown in listing 1.

```
CLR   TYA           ; Y comes in with 0; 0 means a blank
      LDY #39       ; 40 characters/line (0 thru 39)
LOOP  STA (88), Y   ; Fill from end to beginning
      DEY           ; Done?
      BPL LOOP      ; Keep going if not
      JMP NEXT      ; Re-enter the FORTH operating system
```

Listing 1

The CODE definition equivalent to listing 1 would be:

```
HEX                                     (put in hex mode)
CODE CLR                               (define code word)
    98 C,                             (poke in code)
    A0 C, 27 C,
    91 C, 58 C,
    88 C,
    10 C, FB C,
C; DECIMAL                             (end assembly)
```

First, the FORTH system is put into the hexadecimal mode so that opcode values need not be converted to decimal. Next, the word CODE puts the system into an assembly mode and enters the new word CLR into the dictionary as a machine language word. The opcodes are then byte compiled ("C,") into the dictionary. Note that for the final jump to re-enter FORTH, the predefined word NEXT was word compiled (",") into the dictionary. The word C; terminates the assembly process. The system is then restored to the decimal mode.

This method can always be used, but it is very tedious. Each opcode must be looked up, and all relative branches calculated. Besides introducing a great source for error, if a single opcode is added or deleted, it is possible that many jumps must be re-calculated. For this reason, using the assembler is the prescribed method for entering machine language routines.

Unlike the standard assembler which has four fields (the label field, the operation field, the operand field, and the comment field), the FORTH assembler has only three fields. In a FORTH assembler, there is no explicit label field, but there is an implied label field through the use of the assembler constructs IF, and BEGIN, described later. In addition, the remaining three fields in the FORTH assembler are in reversed order (as is standard for the FORTH language). In other words, the operand precedes the operation, and remarks can be embedded anywhere.

In compiling an assembly word, the FORTH assembler ultimately uses either "," or "C," and for this reason assembly mnemonics traditionally end with a comma. valFORTH equivalents are shown in chart 1.

Standard Assembler

```
LDX COUNT
JMP COUNT+1
LDA #3
ADC N
STY TOP,X
INC BOT,Y
STA (TOP,X)
AND (BOT),Y
JMP (POINT)
DEC N+4
DEX
ROL A
```

valFORTH Assembler

```
COUNT LDX,
COUNT 1+ JMP,
# 3 LDA,
N ADC,
TOP ,X STY,
BOT ,Y INC,
TOP X) STA,
BOT )Y AND,
POINT )JMP,
N 4 + DEC,
DEX,
.A ROL,
ROL.A,
```

or

Note: # 9 LDA, = 9 # LDA,
TOP ,X ROL, = ,X TOP ROL, etc.

Chart 1

Converting the program given in listing 1 to FORTH assembly mnemonics we have:

DECIMAL
CODE CLR

```
TYA,
# 39 LDY,
BEGIN,
88 )Y STA,
DEY,
MI UNTIL,
NEXT JMP,
```

```
( TYA )
( LDY #39 )
( LOOP )
( STA (88),Y )
( DEY )
( BPL LOOP )
( JMP NEXT )
```

C;

In the above example, a BEGIN ... UNTIL, clause (described in the next section) is used. By using this structure, no labels are necessary and positive logic is used rather than negative logic (i.e., "repeat until minus" instead of "if NOT minus, then repeat"). Note that the FORTH assembler compiles exactly the same machine code as the standard assembler, it simply makes the assembly coding easier.

Control Structures

Allowing labels within assembly language programming would make the FORTH assembler needlessly long and slow. To get around the problem of test branching, the ValFORTH assembler has a very powerful set of control structures similar to those found in high level FORTH.

The IF,...ENDIF, and IF,...ELSE,...ENDIF, clauses

The IF, construct which handles conditional downward branches has the following two forms:

...code...	...code...
flag IF,	flag IF,
...true code...	...true code...
ENDIF,	ELSE,
...code...	...false code...
	ENDIF,
	...code...

where "flag" is one of the 6502 statuses: NE , EQ , CC , CS , VC , VS , MI , or PL. The following are a few examples of how these are used.

Note: When the FORTH inner interpreter passes control to an assembly language routine, the Y register always contains a zero value and the X register must be preserved as it is used by the FORTH system to maintain the parameter stack. See the section on parameter passing for more information.

```

; Code routine for 1+
ONEPL INC 0,X      ; increment low byte of 16 bit value
      BNE THERE   ; carry out of low?
      INC 1,X     ; increment high byte if so
THERE JMP NEXT    ; re-enter FORTH system

```

Now in ValFORTH assembly language:

```

CODE ONEPL      (define word)
  0 ,X INC,     (increment low byte)
  EQ IF,        (if result was zero,)
    1 ,X INC,   (then bump the high byte)
  ENDIF,
  NEXT JMP,     (exit to FORTH)
C;

```

Note: In the following example, CONIN is assumed to be predefined.

```

; Input routine
INPUT JSR CONIN      ; Go get character, comes back in A
      CMP #$0D       ; Is it a carriage return?
      BNE INP1        ; If not, do something else
      ...code1...     ; execute code for carriage return
      JMP INP2        ; do not execute "normal" code
INP1  ...code2...     ; execute code for normal keys
INP2  ...code3...     ; execute code more common code
      JMP NEXT        ; re-enter FORTH system

```

The equivalent valFORTH version would be:

```

HEX
CODE INPUT
  CONIN JSR      (Get character      )
  # OD CMP,     (carriage return?  )
  EQ IF,        (If so, then       )
  ...code1...   (execute c/r code   )
  ELSE,         (otherwise          )
  ...code2...   (execute normal code)
  ENDIF,
  ...code3...
  NEXT JMP,     (re-enter FORTH system )
C; DECIMAL

```

The BEGIN,...UNTIL, clause

Another useful structure is the BEGIN,...UNTIL, construct which allows for post-testing indefinite looping. The BEGIN,...UNTIL, construct has the following form:

```

...code1...
BEGIN,          code2 is repeatedly
...code2...     executed until "flag"
flag UNTIL,     is true.
...code3...

```

The following 6502 routine waits until a carriage return has been typed.

```

; WAIT until c/r
WAIT JSR CONIN      ; Go get a character, comes back in A
      CMP #$0D       ; Is it a carriage return?
      BNE WAIT        ; Ask again if not
      JMP NEXT        ; Return to FORTH

```

Using the BEGIN, clause, this becomes

```

NEXT
CODE WAIT          (Code name WAIT )
  BEGIN,          (Begin waiting  )
  CONIN JSR,      (Get a character )
  # OD CMP,       (Carriage return?)
  EQ UNTIL,       (loop up until so)
  NEXT JMP,
C; DECIMAL

```

The BEGIN,...WHILE,...REPEAT, clause

In the valFORTH assembler, there is another valuable control structure. It is the BEGIN,...WHILE,...REPEAT, structure. The WHILE, clause allows pre-testing indefinite loops to be easily programmed. It takes the form:

...code1...	
BEGIN,	Code2 and code3 are repeatedly
...code2...	executed until "flag" become
flag WHILE,	false, at which time program
...code3...	control proceeds to code4.
REPEAT,	
...code4...	

A common example of the WHILE, clause is getting a line of input text terminated by a carriage return.

;	Get line of text	(note: Y=0 on entry always)	
GETLN	JSR CONIN	;	Get one character
	CMP #\$0D	;	C/R terminates input
	BEQ GETL1	;	If not a C/R then
	STA BFFR,Y	;	store the character
	INY	;	Bump buffer pointer
	JMP GETLN	;	Go back for more
GETL1	JMP NEXT	;	Exit to FORTH

Using the WHILE, clause in valFORTH, we have:

HEX			
CODE	GETLN		
	BEGIN,		
	CONIN JSR,	(Get a character)
	# 0D CMP,	(Carriage return?)
	NE WHILE,	(If not,)
	BFFR ,Y STA,	(then store the character)
	INY,	(and bump the pointer)
	REPEAT,	(Repeat all of the above)
	NEXT JMP,		
C;	DECIMAL		

The BEGIN,...AGAIN, clause

The final control structure is the BEGIN,...AGAIN, structure. This structure allows the use of unconditional looping in assembly language routines. Although its use is rare, it can reduce code size considerably. It takes the following form:

```

...code1...
BEGIN,                                     Repeatedly execute code2
...code2...                               and code4 until "flag"
flag IF,                                 becomes true, in which
...code3...                               case, program execution
re-entry-point JMP,                       continues with code3 and
ENDIF,                                   a system re-entry made.
...code4...
AGAIN, C;

```

The best example of the AGAIN, clause is in the coding of the CMOVE routine:

```

; Byte at a time front end memory move
CMOVE LDA #3                               ; Get top three stack items
      JSR SETUP                             ; Move them to N scratch area
CMOV1 CPY N                               ; Time to decrement COUNT high?
      BNE CMOV2                             ; Nope
      DEC N+1                               ; Yes, so do it
      BPL CMOV2                             ; Bypass exit if not done
      JMP NEXT                             ; Exit to FORTH system
CMOV2 LDA (N+4),Y                         ; Get byte to move
      STA (N+2),Y                           ; Move it!
      INY                                   ; Bump byte pointer
      BNE CMOV1                             ; Keep going until ready to
      INC N+5                               ; bump high bytes of both
      INC N+3                               ; "to" and "from" addresses
      JMP CMOV1                             ; Do it all again

```

Using the AGAIN, clause, this becomes:

```

CODE CMOVE
  # 3 LDA,                                (Prepare for memory move)
  SETUP JSR,
  BEGIN,
    BEGIN,                               (Start the process)
    N CPY,                               (done?)
    EQ IF,
    N 1+ DEC,                             (Maybe, keep checking)
    MI IF,
    NEXT JMP,                             (Re-enter FORTH system)
    ENDIF,
    ENDIF,
    N 4 + )Y LDA,                         (Get byte to copy)
    N 2+ )Y STA,                         (Store in new location)
    INY,                                 (Bump pointer)
    EQ UNTIL,
    N 5 + INC,                             (Bump addresses)
    N 3 + INC,
  AGAIN,                                 (Do it all again)
C; DECIMAL

```

Parameter Passing

One of the most useful features of the FORTH language is its ability to use a parameter stack for passing values from one word to another. For assembly language routines to really be useful in the FORTH system, there must be some facility for these routines to access this stack. Likewise, there should be some way in which to access the return stack as well. This section details exactly how to make the best use of both stacks.

Since the FORTH system maintains dual stacks and the 6502 supports only one, it is necessary to simulate one of the stacks. For ease of stack manipulation, the parameter is simulated; the return stack uses the hardware stack of the microprocessor.

The simulated stack uses the 0-page,X addressing mode of the 6502. For example, the following statements show how the parameter stack is organized.

LDA 0,X	Low byte of item on top of stack
INC 1,X	High byte of top item
ADC 2,X	Low byte of item second on stack
EOR 3,X	High byte of 20S
RNL 4,X	Low byte of item third on stack
AND 5,X	
...	etc.

In high level FORTH, the word DROP drops (or pops) the top value from the stack. The code definition for DROP is:

```
CODE DROP      INX, INX, NEXT JMP, C;
```

In the same way, values can be "pushed" to the stack. Note that the X register must be preserved between FORTH words or the parameter stack is lost! Thus if the X register is needed in a code definition, it must be saved upon entry to the routine and restored before returning to the FORTH system. The special location XSAVE is reserved for this: (The word XSAVE has been defined as a FORTH constant.)

STX XSAVE	Save the X register
LDX XSAVE	Restore the X register

In all the examples given so far, the code definitions have re-entered the FORTH system through the normal re-entry point called NEXT. The following is a complete description of all possible re-entry points: (In all of the following code examples, standard 6502 assembler format has been used for ease of comprehension. All valFORTH assembler equivalents can be found in appendix A.)

The NEXT re-entry point

The NEXT routine transfers control to the next FORTH word to be executed. All FORTH words eventually come through the NEXT routine.

Likewise, all other re-entry points come through NEXT once they have completed their special tasks. The next routine is typically used by words

The NEXT re-entry point (cont'd)

such as 1- which do not modify the number of arguments on the stack. The word NEXT is defined as a FORTH constant. NXT, is an abbreviation for NEXT JMP, .

```
Example:      ; 1- routine
              ONEM LDA 0,X      ; Borrow from low byte?
              BNE ONE1         ; If not, ignore correction
              DEC 1,X          ; Decrement high byte
              ONE1 DEC 0,X      ; Now do the low
              JMP NEXT         ; Re-enter FORTH
```

Listing 2

The PUSH re-entry point:

The PUSH routine pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accumulator. The X register is automatically decremented twice for the two bytes. This routine is typically used for words such as OVER or DUP which leave one more argument than they expect. The word PUSH has been defined as a FORTH constant. PSH, is an abbreviation for PUSH JMP, .

```
Example:      ; DUP routine
              DUP  LDA 0,X      ; Get low byte of TOS
              PHA              ; Push it
              LDA 1,X          ; Put high byte in A
              JMP PUSH         ; Put it on the P-stack
```

Listing 3

The PUT re-entry point:

The PUT routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator. This is used by words such as ROT or SWAP which do not change the number of values on the stack. The word PUT has been defined as a FORTH constant. PUT, is an abbreviation for PUT JMP, .

```
Example:      ; SWAP routine
              SWAP LDA 2,X      ; Low byte of 2nd value
              PHA              ; Save it
              LDA 0,X          ; Put low byte of TOS
              STA 2,X          ; into low byte of 2OS
              LDA 3,X          ; Hold high byte of 2OS
              LDY 1,X          ; Put high byte of TOS
              STY 3,X          ; into high byte of 2OS
              JMP PUT          ; Replace TOS no
```

Listing 4

The PUSHOA re-entry point

The PUSHOA re-entry point pushes the 8 bit unsigned value in the accumulator as a 16 bit value with the upper 8 bits zeroed. This word is very commonly used by words which leave a boolean flag on the parameter stack such as ?TERMINAL. The word PUSHOA has been defined as a FORTH constant. PSHA, is an abbreviation for PUSHOA JMP, .

```
Example: ; ?TERMINAL routine
          QTERM LDA $D01F      ; Read Atari CONSOLE keys
          EOR #7               ; Anything pressed?
          BEQ QT1              ; If not, go push false
          INY                  ; Else push a true
          QT1 TYA               ; Put Y (0 or 1) in A
          JMP PUSHOA           ; Go push the result
```

Listing 5

The PUTOA re-entry point:

The PUTOA routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero. This is used by words such as C@ which simply replace their arguments on the stack. The word PUTOA is defined as a FORTH constant. PUTA, is an abbreviation for PUTOA JMP, .

```
Example: ; Byte fetch
          CFCH LDA (0,X)       ; Load byte indirectly
          JMP PUTOA            ; Replace the address
                                ; with the contents
```

Listing 6

The BINARY re-entry point

The BINARY re-entry point drops the value on top of the parameter stack and then performs the PUT operation described above. This word is commonly used by words such as XOR which use one more argument than they leave. The word BINARY has been defined as a FORTH constant.

```
Example: ; Exclusive or TOS with 2OS
          XOR LDA 0,X          ; Get low byte of top value
          EOR 2,X              ; XOR it with low of 2OS
          PHA                  ; Save it
          LDA 1,X              ; Now do same for high bytes
          EOR 3,X              ; Result in A
          JMP BINARY           ; Go DROP , PUT
```

Listing 7

POP and POPTWO re-entry points

The POP and POPTWO re-entry points are used when values must be dropped from the parameter stack. POP performs a DROP, while POPTWO performs a 2DROP. Most words which can use BINARY can use POP. The words POP and POPTWO have been defined as FORTH constants. POP, is an abbreviation for POP JMP, and POP2, is an abbreviation for POPTWO JMP, .

Examples: ; Another XOR routine

```

XOR  LDA 0,X      ; Get low byte
      EOR 2,X      ; XOR with other low byte
      STA 2,X      ; Put directly on stack
      LDA 1,X      ; Do the same for high bytes
      EOR 3,X
      STA 3,X
      JMP POP      ; Remove unneeded TOS item

```

Listing 8

```

; C! routine
CSTR LDA 2,X      ; Get byte to store
      STA (0,X)    ; Store it!
      JMP POPTWO   ; Drop byte and address

```

Listing 9

The SETUP routine

A very useful routine in the FORTH system is the code routine SETUP. On the 6502, 0-page addressing is typically faster than absolute addressing. Also, some instructions, such as indirect-indexed addressing, can use only 0-page addresses. The SETUP routine allows the assembly language programmer to transfer up to four stack values to a scratch pad in the 0-page for these operations. The predefined name for this area is N. The calling sequence for the SETUP routine is:

```

LDA #num      ; Move "num" values to N, ("num" = 1-4)
JSR SETUP     ; then drop "num" values from the stack

```

The SETUP routine moves one to four values to the N scratch area and drops all values moved from the parameter stack. These values are stored in the following order:

```

LDA N          ; Low byte of value that was TOS
EOR N+1        ; High byte           ( N 1+ EOR, )
ADC N+2        ; Low byte of value that was 2OS
STY N+3        ; High byte           ( N 3 + STY, )
INC N+4        ; Low byte of 3OS      ( N 4 + INC, )
...
DEC N+7        ; High byte of value that was 4OS

```

Words such as CMOVE and FILL which use indirect-indexed addressing typically use the SETUP routine (see the BEGIN,...AGAIN, example). The word SETUP has been defined as a FORTH constant.

Return stack manipulation

The FORTH return stack is implemented as the normal 6502 hardware stack. To push and pop values, the 6502 stack instructions PHA and PLA can be used. Sometimes it is also necessary to manipulate the data on the return stack (such as for DO looping). Using the normal stack operations to do this can be tedious. Using indexed addressing, the return stack can be manipulated in the same manner as the parameter stack.

Examples: ; >R routine
 TOR LDA 1,X ; Pick up high byte
 PHA ; Push it to R
 LDA 0,X ; Now do the low byte
 PHA ; It's done!
 JMP POP ; Now, "lose" TOS

Listing 10

```

; 3rd loop index (I , I' , J , ... K )
K   STX XSAVE ; Save P-stack pointer
    TSX       ; Get R-stack pointer
    LDA $109,X ; 101-102,...,109-10A, (L-H)
    PHA       ; Push low byte of 3rd item
    LDA $10A,X ; A now has high byte
    LDX XSAVE ; Restore P-stack pointer
    JMP PUSH  ; Push the index

```

Listing 11

Machine Language Subroutines in valFORTH

When coding in assembly language, it is often useful to be able to make subroutine calls for often used operations. Using CODE makes it possible to do this, but it is not recommended. The following subroutine uses CODE.

```

CODE S1+      ( Subroutine 1+      )
  0 ,X INC,   (   INC 0,X          )
  EQ IF,      (   BNE *+4          )
  1 ,X INC,   (   INC 1,X          )
  ENDIF,      (                    )
  RTS,        (   RTS              )
C;

```

This subroutine could now be used in assembly language routines in the following way:

```

CODE 1+      (Another 1+ routine )
  ' S1+ JSR,  (   JSR S1+         )
  NEXT JMP,   (   JMP NEXT        )
C;

```

This works fine, but there is one slight problem. If the user types S1+ as a command (i.e., it is not called, but executed) the FORTH system will "crash" when the RTS statement is encountered. This is because FORTH does not call its words, but jumps to them. For this reason, CODE is not used. A word which acts like CODE but protects the system is needed.

In the code for 1+ above, it was necessary to ' (tick) the subroutine to find its address. It would be desirable if we could simply type its name and have it return its address (just as NEXT and PUSH do). This is possible. The word SUBROUTINE below allows this (note that this word is not automatically loaded with the assembler, it must be typed in by the user).

```

: SUBROUTINE          (new word SUBROUTINE )
  0 VARIABLE          (is like a VARIABLE )
  -2 ALLOT            (discard the value of 0 )
  [COMPILE] ASSEMBLER (Put into assembly mode )
  ?EXEC !CSP ;        (Set/check for errors )

```

The word SUBROUTINE can be used in the same way CODE is except that SUBROUTINES end with an RTS instruction while CODE routines must end with a jump to a re-entry point. When the word defined using SUBROUTINE is executed, the entry point to the routine is left on the stack similar to the way in which a word defined using VARIABLE leaves an address. The following is an example of subroutine usage.

```

SUBROUTINE 2'SCOMP      (Two's complement )
  SEC,                  (routine )
  0 # LDA,
  0 ,X SBC,              (i.e., TOS => - TOS)
  0 ,X STA,
  0 # LDA,
  1 ,X SBC,
  1 ,X STA,
  RTS,
C;

```

It can now be used as such:

```

CODE ABS                (Take abs. value of TOS )
  1 ,X LDA,              (Is TOS < 0 ? )
  MI IF,
    2'SCOMP JSR,         (If so, TOS => -TOS )
  ENDIF,
  NEXT JMP,              (Exit to FORTH system )

```

When the new word 2'SCOMP is executed directly, it leaves its address on the stack. When it is called by a subroutine, it performs a two's complement on the top stack value. This dual type of execution allows safe access to assembly language subroutines.

Macro Assemblies in valFORTH

FORTH assemblers use a reversed form of notation so that all the benefits of the standard FORTH system are available. In other words, anything that can be done in FORTH can be done during assembly time in a code definition. This is because all of the assembler opcodes are actually FORTH words which take arguments from the parameter stack. Thus

NEXT JMP,

actually puts the address of the NEXT routine (NEXT is a FORTH constant) onto the parameter stack. The word JMP, then compiles the address into the dictionary. Here is a simplified definition (it does not test for indirect jumping) for the JMP, opcode:

```

HEX
: JMP,          (address --- )
  4C C,        (compile in JMP opcode )
  ,           (compile in the address )
; DECIMAL

```

All assembly words are designed in this fashion. Thus the necessity for operands to precede opcodes becomes clear. This allows the use of complex assembly time calculations that no ordinary assembler would ever support (e.g. no standard 6502 assembler would allow the use of the SIN function for generating a data table).

Most assemblers do allow the use of the basic operations; + , - , * , / , and &. These are easily used in the valFORTH system:

```

LDA #COUNT&$FF      COUNT FF AND # LDA,
LDY #NAME/$100        COUNT 100 / # LDY,
EOR N+6               N      6 +   EOR,
LDX #"A"+$80          ASCII A 80 + # LDX,
etc.

```

The looping structures IF, and BEGIN, each leave two values on the stack during assembly time. The first is a branch address, the second is an identification code. When ENDIF, is executed, it checks the identification code to verify that structures have not been illegally interleaved (i.e., BEGIN, ... ENDIF,). If everything checks out, ENDIF, then calculates the branch offset required by the IF, clause, otherwise an error is reported. The BEGIN, clause functions in the same manner. Thus, the words IF, and BEGIN, are predefined macro instructions in the valFORTH assembler.

The fact that a FORTH assembler is nothing but a collection of words means that the assembler, like the FORTH language itself, is extensible. In other words, macro assemblies can easily be performed by defining new assembler directives. Take the following code extract which outputs a text string:


```

...code...           ; ...
JSR CRLF             ; Skip to next line
JSR PRTXT            ; Call print routine
.BYTE 11,'valFORTH 1.' ; String to output
LDA REL              ; Get release number
JSR PRTNM            ; Print the number
JSR CRLF             ; Issue c/r
...code...           ; ...

```

This code prints out the string "valFORTH 1.x" where "x" is the release number. Note that the routine PRTXT does not exist, it is simply used here for example purposes. The PRTXT routine "pops" the return address which points to the output string, picks up the length byte and adds it to the return address. The return address, which now points to the LDA instruction is "pushed" back onto the stack. The PRTXT routine still has a pointer to the string which it then prints out. Finally, it does an RTS and returns control to the calling program. The release number is then printed out.

Assuming that the PRTXT routine is used quite often, it would be desirable to make it an assembler macro. A word which automatically assembles in the subroutine call to PRTXT and then assembles in a user specified string would be quite handy. In valFORTH, this is easily accomplished:

```

ASSEMBLER DEFINITIONS (This is an assembler word)
HEX                   (Put system in base 16)
: PRINT"              (Command form: PRINT" text")
  20 C, PRTXT ,       (compile in JSR PRTXT)
  22 WORD              (Now the string upto ")
  HERE C@ 1+ ALLOT     (Bump dictionary pointer)
; DECIMAL              (all done,)
IMMEDIATE             (make word execute even at)
FORTH DEFINITIONS     (compile time.)

```

This word could now be used in ValFORTH assemblies in the following manner:

```

...code...
CRLF JSR,              (Skip to next line)
PRINT" ValFORTH 1."    (Print out string)
REL LDA,               (Get release number)
PRTNM JSR,             (Go print it)
CRLF JSR,              (Skip to next line)
...code...

```

Using the newly defined PRINT" macro, strings no longer need to be counted, and since there is less text to enter, typing errors are reduced. Other useful macros which could be designed are words which allow conditional assembly or automatically set up IOCB blocks for Atari operating system calls. Experienced assembly language programmers typically have a set of often used routines defined as macro instructions for quicker program development.

Compatability With Other Popular Assemblers

There are several other versions of FORTH out which have 6502 assemblers. The two major versions are the Forth Interest Group's written by William Ragsdale, and the version put out by the Atari Program Exchange written by Patrick Mullarky. The valFORTH assembler is a superset of both of these fine assemblers and is fully compatible with both versions.

Although not stated previously in the documentation, there are several ways in which to implement the IF, , WHILE, and UNTIL, structures. The valFORTH assembler was designed with transportability in mind. Although the recommended method is the valFORTH version, each of the following may be used.

valFORTH	Fig version	APX version
EQ IF,	O= IF,	IFEQ,
NE IF,	O= NOT IF,	IFNE,
CS IF,	CS IF,	IFCS,
CC IF,	CS NOT IF,	IFCC,
VS IF,	-----	IFVS,
VC IF,	-----	IFVC,
MI IF,	O< IF,	IFMI,
PL IF,	O< NOT IF,	IFPL,
EQ WHILE,	-----	-----
NE WHILE,	-----	-----
CC WHILE,	-----	-----
CC WHILE,	-----	-----
VS WHILE,	-----	-----
VC WHILE,	-----	-----
MI WHILE,	-----	-----
PL WHILE,	-----	-----
EQ UNTIL,	O= UNTIL,	O= UNTIL,
NE UNTIL,	O= NOT UNTIL,	O= NOT UNTIL,
CS UNTIL,	CS UNTIL,	-----
CC UNTIL,	CS NOT UNTIL,	-----
VS UNTIL,	-----	-----
VC UNTIL,	-----	-----
MI UNTIL,	O< UNTIL,	-----
PL UNTIL,	O< NOT UNTIL,	-----

Chart 2

In all versions, the word END, is synonymous with the word UNTIL,. Likewise, THEN, is synonymous with ENDIF,.

In the valFORTH and Fig assemblers, compiler security is performed to give added protection to the user against assembly errors. To accomplish this, the word C; or its synonym END-CODE is used to terminate the assembly word and perform the check. To remain compatible with APX FORTH, C; is not required in this release of valFORTH. However, it is strongly recommended that C; be used. Although C; and END-CODE are identical, C; is used in-house at Valpar for brevity. (Note that in later releases of valFORTH, C; will become mandatory).

There are several ways in which the indirect jump in the 6502 architecture is implemented in FORTH assemblers. The valFORTH assembler supports three common versions. Thus,

```
                                JMP (VECTOR)
can be:
                                VECTOR    )JMP,
                                VECTOR    ) JMP,
or   VECTOR    JMP(),
```

It is recommended that the first version be used.

It must be remembered that valFORTH's additional constructs may not be recognized by assemblers available from other vendors. If assembly listings are to be published for general 6502 FORTH users, it is suggested that valFORTH's advanced features not be used so that novice programmers can still make use of valuable pieces of code.

Appendix AvalFORTH Code Equivalents

This appendix gives the valFORTH assembly code for all 6502 code listings which are marked. Although listing 1 has already been translated to valFORTH assembly code, it is reproduced here for completeness.

Listing 1

DECIMAL	
CODE CLR	
TYA,	(Move a blank [0] into A)
# 39 LDY,	(Move count into Y)
BEGIN,	(Start looping)
88)Y STA,	(Move in a blank)
DEY,	(decrement pointer)
MI UNTIL,	(Go until count < 0)
NEXT JMP,	(Do a normal re-entry)
C;	

Listing 2

CODE 1-	(Decrement 16 bit value)
0 ,X LDA,	(Get the low byte)
NE IF,	(If a borrow will occur,)
1 ,X DEC,	(then borrow from high...)
ENDIF,	
0 ,X DEC,	(Decrement low)
NEXT JMP,	(Re-enter FORTH)
C;	

Listing 3

CODE DUP	(Duplicate TOS)
0 ,X LDA,	(Get low byte)
PHA,	(Set up for PUSH)
1 ,X LDA,	(Put high in Accumulator)
PUSH JMP,	(Push 16 bit value)
C;	

Listing 4

CODE SWAP	(Exchange top stack items)
2 ,X LDA,	(Get low byte of 20S)
PHA,	(Save it)
0 ,X LDA,	(Put low byte of TOS)
2 ,X STA,	(into low byte of 20S)
3 ,X LDA,	(Save high byte of 20S)
1 ,X LDY,	(Put high byte of TOS)
3 ,X STY,	(into high byte of 20S)
PUT JMP,	(Put old 20S into TOS)
C;	

Listing 5

HEX	
CODE ?TERMINAL	(Any console key pressed?)
DO1F LDA,	(Load status byte)
# 7 EOR,	(Any low bits reset?)
NE IF,	(If so,)
INY,	(then leave a true value)
ENDIF,	
TYA,	(Put true or false into A)
PUSHOA JMP,	(Push to parameter stack)
C; DECIMAL	

Listing 6

CODE C@	(Byte fetch routine)
0 X) LDA,	(Load from address on TOS)
PUTOA JMP,	(Push byte value)
C;	

Listing 7

CODE XOR	(One example of XOR)
0 ,X LDA,	(Get low byte of TOS)
2 ,X EOR,	(Exclusive or it with 20S)
PHA,	(Push low result)
1 ,X LDA,	(Get high byte of TOS)
3 ,X EOR,	(XOR it with high of 20S)
BINARY JMP,	(Drop TOS and replace 20S)
C;	

Listing 8

CODE XOR	(Another exclusive or)
0 ,X LDA,	(Get low byte of TOS)
2 ,X EOR,	(XOR with low of 20S)
2 ,X STA,	(Put in low of 20S)
1 ,X LDA,	(Get high byte of TOS)
3 ,X EOR,	(XOR with high of 20S)
3 ,X STA,	(Put in high of 20S)
POP JMP,	(Drop TOS)
C;	

Listing 9

CODE C!	(Byte store routine)
2 ,X LDA,	(Pick up byte to store)
0)X STA,	(Indirectly store it)
POPTWO JMP,	(Drop address and byte)
C;	

Listing 10

CODE >R	(Transfer TOS to R-stack)
1 ,X LDA,	(Pick up high of TOS)
PHA,	(Put on R-stack)
0 ,X LDA,	(Pick up low of TOS)
PHA,	(Put on R-stack)
POP JMP,	(Lose top stack item)
C;	

Listing 11

HEX	
CODE K	(3rd inner D0 loop index)
XSAVE STX,	(Save P-stack pointer)
TSX,	(Pick up R-stack pointer)
109 ,X LDA,	(Pick up low byte of value)
PHA,	(Save it)
10A ,X LDA	(Put high byte of value in A)
XSAVE LDX,	(Restore P-stack pointer)
PUSH JMP,	(Push 16 bit index value)
C; DECIMAL	

Appendix BQuick Reference Chart

valFORTH 6502 Assembly Words

ASSEMBLER (---)

Calls up the assembler vocabulary for subsequent assembly language programming.

CODE cccc (---)

Enters the new word "cccc" into the dictionary as machine language word and calls up the assembler vocabulary for subsequent assembly language programming. CODE also sets the system up for security checking.

C; (---)

Terminates an assembly language definition by performing a security check and setting the CONTEXT vocabulary to the same as the CURRENT vocabulary.

END-CODE (---)

A commonly used synonym for the word C; above. The word C; is recommended over END-CODE.

SUBROUTINE cccc (---)

Enters the new word "cccc" into the dictionary as machine language subroutine and calls up the assembler vocabulary for subsequent assembly language programming. SUBROUTINE also sets the system up for security checking.

;CODE (---)

When the assembler is loaded, puts the system into the assembler vocabulary for subsequent assembly language programming. See main glossary for further explanation.

Control Structures

IF, (flag --- addr 2)

Begins a machine language control structure based on the 6502 status flag on top of the stack. Leaves an address and a security check value for the ELSE, or ENDIF, clauses below. "flag" can be EQ , NE , CC , CS , VC , VS , MI , or PL . Command forms:

...flag..IF,..if-true..ENDIF,...all...
 ...flag..IF,..if-true..ELSE,..if-false..ENDIF,...all...

ELSE, (addr 2 --- addr 3)

Used in an IF, clause to allow for execution of code only if IF, clause is false. If the IF, clause is true, this code is bypassed. See IF, above for command form.

ENDIF, (addr 2/3 ---)

Used to terminate an IF, control structure clause. Additionally, ENDIF, resolves all forward references. See IF, above for command form.

BEGIN, (--- addr 1)

Begins machine language control structures of the following forms:

...BEGIN,...AGAIN,...
 ...BEGIN,...flag..UNTIL,...
 ...BEGIN,...flag..WHILE,..while-true..REPEAT,...

where "flag" is one of the 6502 statuses: EQ , NE , CC , CS , VC , VS , MI , and PL . See the very similar BEGIN in the main glossary for additional information.

UNTIL, (addr 1 flag ---)

Used to terminate a post-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is false. See BEGIN, above for more information.

WHILE, (addr 1 flag --- addr 4)

Used to begin a pre-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is true. See BEGIN, above for command format.

REPEAT, (addr 4 ---)

Used to terminate a pre-testing BEGIN,..WHILE, clause. Additionally, REPEAT, resolves all forward addresses of the current WHILE, clause. See BEGIN, above.

AGAIN, (addr 1 ---)

Used to terminate an unconditional BEGIN, clause. Execution cannot exit this loop unless a JMP, instruction is used. See BEGIN, clause for more information.

Parameter Passing

NEXT (--- addr)

Transfers control to the next FORTH word to be executed. The parameter stack is left unchanged.

PUSH (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accumulator.

PUSHOA (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found in the accumulator and whose high byte is zero.

PUT (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator.

PUTOA (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero.

BINARY (--- addr)

Drops the top value of the parameter stack and then performs a PUT operation described above.

POP and POPTWO (--- addr)

POP drops one value from the parameter stack. POPTWO drops two values from the parameter stack.

SETUP (--- addr)

Moves one to four values to the N scratch area in the zero page and drops all values moved from the parameter stack.

N (--- addr)

Points to a nine-byte scratch area in the zero page beginning at N-1 and going to N+7. Typically used by words which use indirect-indexed addressing where addresses must be stored in the zero page. See SETUP above.

Opcodes

(---)

ADC,	AND,	ASL,	BIT,	BRK,	CLC,	CLD,	CLI,
CLV,	CMP,	CPX,	CPY,	DEC,	DEX,	DEY,	EOR,
INC,	INX,	INY,	JSR,	JMP,	LDA,	LDX,	LDY,
LSR,	NOP,	ORA,	PHA,	PHP,	PLA,	PLP,	ROL,
ROR,	RTI,	RTS,	SBC,	SEC,	SED,	SEI,	STA,
STX,	TAX,	TAY,	TSX,	TXA,	TXS,	TYA,	

Aliases

NXT,	=	NEXT JMP,
PSH,	=	PUSH JMP,
PUT,	=	PUT JMP,
PSHA,	=	PUSHOA JMP,
PUTA,	=	PUTOA JMP,
POP,	=	POP JMP,
POP2,	=	POPTWO JMP,
XL,	=	XSAVE LDX,
XS,	=	XSAVE STX,
THEN,	=	ENDIF,
END,	=	UNTIL,

VII. valFORTH 1.1 SUPPLIED SOURCE LISTING

Screen: 30

```

0 ( Auto command )
1
2 BASE @ HEX
3
4 : ZAP          ( addr # -- )
5   -DUP
6   IF 0+S
7     DO D20A C@ 7F AND I C!
8     LOOP
9   ELSE DROP
10  ENDIF ;
11
12 : -WAIT          ( -- )
13   BEGIN ?TERMINAL NOT UNTIL ;
14
15 DCX              ==>

```

Screen: 33

```

0 ( Auto command )
1
2 : QUEST2          ( -- n )
3   ." Format and save:  "
4   ." press OPTION" CR
5   ." Just save:      "
6   ." press SELECT" CR CR
7   WAIT ?TERMINAL -WAIT
8   ." Prepare disk -- "
9   ." press START"
10  WAIT -WAIT ;
11 : CSV            ( -- )
12   ." Prepare cassette "
13   ." (play/record) -- " CR
14   ." press START" CR
15  WAIT CSAVE -WAIT ;  -->

```

Screen: 31

```

0 ( Auto command )
1
2 : BEHEAD          ( -- )
3   0 > R CR ." Now protecting..."
4   CR VOC-LINK @
5   BEGIN
6     DUP 2- @
7     BEGIN
8       DUP 1+ OVER R> 1+ > R
9       R 15 MOD NOT IF ." ." ENDIF
10      R 495 MOD NOT IF CR ENDIF
11      C@ 63 AND WIDTH @ MIN 1-
12      ZAP PFA LFA @ DUP NOT
13      UNTIL
14      DROP @ DUP NOT
15      UNTIL R> 2DROP ;  -->

```

Screen: 34

```

0 ( Auto command )
1
2 : DSV
3   QUEST2
4   4 =
5   IF
6     1 (FMT) 1 (>)
7     IF
8       CR ." Format error"
9     ELSE
10      DODISK
11    ENDIF
12    ELSE
13      DODISK
14    ENDIF CR ;
15  ==>

```

Screen: 32

```

0 ( Auto command )
1
2 : QUEST          ( -- )
3   CR
4   ." Save on disk:  "
5   ." press OPTION" CR
6   ." Save on cassette: "
7   ." press SELECT" CR
8   ." Exit:        "
9   ." press START" CR CR ;
10
11 : DODISK          ( -- )
12   (SAVE) ' SAVE 32 + @EX
13   741 @ 128 - 1 1 R/W ;
14
15  ==>

```

Screen: 35

```

0 ( Auto command )
1
2 : DECIS          ( -- )
3   BEGIN
4     QUEST WAIT ?TERMINAL -WAIT
5     DUP 1 =
6     IF DROP 1
7     ELSE
8       2 =
9     IF
10      CSV
11    ELSE
12      DSV
13    ENDIF @
14    ENDIF
15    UNTIL ;  -->

```

Screen: 36

```

0 ( Auto command )
1
2 : AUTO ( -- )
3 [COMPILE] ' CR
4 ." Auto? Y/N " KEY 89 = CR
5 IF
6 CFA ' ABORT 6 + !
7 ' COLD CFA ' ABORT 8 + !
8 -1 26 +ORIGIN !
9 ' ZAP NFA DP !
10 BEHEAD DECIS ABORT
11 ELSE
12 DROP ." Auto aborted..." CR
13 ENDIF ; BASE !
14
15

```

Screen: 37

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 38

```

0 ( Text output: S: P: )
1
2 BASE @ HEX
3
4 : S: ( f -- )
5 PFLAG @ SWAP
6 IF 1 OR ELSE FE AND ENDIF
7 PFLAG ! ;
8
9 : P: ( f -- )
10 PFLAG @ SWAP
11 IF 2 OR ELSE FD AND ENDIF
12 PFLAG ! ;
13
14
15 ==>

```

Screen: 39

```

0 ( Text output: BEEP ASCII )
1
2 : BEEP ( -- )
3 0C0 0
4 DO
5 08 0D01F C! 6 0 DO LOOP
6 00 0D01F C! 6 0 DO LOOP
7 LOOP ;
8
9 : ASCII ( ccc, -- <b> )
10 BL WORD
11 HERE 1+ C@
12 STATE @
13 IF
14 COMPILER CLIT C,
15 ENDIF ; IMMEDIATE -->

```

Screen: 40

```

0 ( Text output: EJECT LISTS )
1 DCX
2
3 : EJECT ( -- )
4 12 EMIT ;
5
6 : LISTS ( s # -- )
7 0 <ROT 0+S
8 DO
9 CR I LIST
10 1+ DUP 3 MOD 0=
11 IF EJECT ENDIF
12 ?EXIT
13 LOOP
14 DROP ;
15 ==>

```

Screen: 41

```

0 ( Text output: PLISTS PLIST )
1
2 : PLISTS ( s # -- )
3 PFLAG @ <ROT
4 ON P:
5 LISTS
6 CR PFLAG ! ;
7
8 : PLIST ( s -- )
9 1 PLISTS ;
10
11
12
13
14
15 BASE !

```

Screen: 42

```

0 ( Debug: B? [FREE] FREE )
1 BASE @ DCX
2 '( S: )( 19 KLOAD )
3 HEX
4
5 : B? ( -- )
6 BASE @ DUP ( Display )
7 DECIMAL . ( current )
8 BASE ! ; ( radix )
9
10 : (FREE) ( -- n )
11 2E5 @ PAD - ;
12
13 : FREE ( -- )
14 (FREE) U. ." bytes" CR ;
15 ==>

```

Screen: 45

```

0 ( Debug: CDUMP )
1
2 ( Character dump routine )
3
4 : CDUMP ( a # -- )
5 PFLAG @ (ROT OFF P:
6 OVER + SWAP
7 DO
8 CR I 5 U.R
9 SPACE I DUP 10 + SWAP
10 DO
11 I C@ SPEMIT
12 LOOP
13 ?EXIT
14 10 /LOOP
15 CR PFLAG ! ; -->

```

Screen: 43

```

0 ( Debug: H. CFALIT )
1
2 : CFALIT ( ccc, -- (b) )
3 STATE @
4 [COMPILE] [
5 [COMPILE] ' CFA
6 SWAP IF [COMPILE] ] ENDIF
7 [COMPILE] LITERAL ;
8 IMMEDIATE
9
10 '( H. --> )( )
11 : H. ( b -- )
12 BASE @ HEX ( Display )
13 SWAP @ ( # in hex )
14 <# # #> TYPE
15 BASE ! ; -->

```

Screen: 46

```

0 ( Stack print: DEPTH )
1
2 : DEPTH ( n -- )
3 S@ @ SP@ - 2/ 1- ;
4
5 CFALIT . VARIABLE X.S
6
7 : XDOTS ( -- )
8 DEPTH
9 IF
10 SP@ 2- S@ @ 2-
11 DO I @ X.S @ EXECUTE
12 -2 +LOOP
13 ELSE
14 ." Stack empty "
15 ENDIF ; ==>

```

Screen: 44

```

0 ( Debug: #DUMP )
1
2 ( memory dump )
3
4 : #DUMP ( a # -- )
5 O+S
6 DO
7 CR I 5 U.R I
8 DUP 8 + SWAP
9 DO
10 I C@ 4 .R
11 LOOP
12 ?EXIT
13 8 /LOOP
14 CR ;
15 ==>

```

Screen: 47

```

0 ( Stack print: .S U.S STACK )
1
2 : .S CFALIT . X.S ! XDOTS ;
3 : U.S CFALIT U. X.S ! XDOTS ;
4
5 : STKPRT
6 CR ." ( " XDOTS ." ) " ;
7
8 : STACK ( f -- )
9 IF
10 CFALIT STKPRT
11 ELSE
12 CFALIT NOOP
13 ENDIF
14 [ ' PROMPT 11 + ]
15 LITERAL ! ; -->

```

Screen: 48

```

0 ( FORTH colon decompiler )
1
2
3 0 VARIABLE .WORD
4
5 : PWORD
6 2+ NFA ID. ;
7
8 : 1BYTE
9 PWORD .WORD @ C@ .
10 1 .WORD +! ;
11
12 : 1WORD
13 PWORD .WORD @ @ .
14 2 .WORD +! ;
15 ==>

```

Screen: 51

```

0 ( FORTH colon decompiler )
1
2 ELSE
3 DUP CFALIT CLIT =
4 IF 1BYTE
5 ELSE
6 DUP CFALIT COMPILE =
7 IF PWORD CR NXT1 PWORD
8 ELSE
9 DUP 4 - @ A922 =
10 IF STG
11 ELSE PWORD ENDIF
12 ENDIF
13 ENDIF
14 ENDIF
15 ENDIF ; -->

```

Screen: 49

```

0 ( FORTH colon decompiler )
1 : NP ( n -- n )
2 DUP CFALIT ;S = OVER
3 CFALIT (;CODE) = OR
4 IF PWORD CR CR PROMPT QUIT
5 ENDIF ?TERMINAL
6 IF DROP PROMPT QUIT ENDIF ;
7
8 : BRNCH
9 PWORD ." to " .WORD @ .WORD @
10 @ + U. 2 .WORD +! ;
11
12 : NXT1
13 .WORD @ U. 2 SPACES
14 .WORD @ @ 2 .WORD +! ;
15 -->

```

Screen: 52

```

0 ( FORTH colon decompiler )
1
2 : ?DOCOL
3 DUP 2- @
4 [ ' : 12 + ] LITERAL -
5 IF ." Primitive pfa dump:"
6 2- @ 18 #DUMP
7 PROMPT QUIT
8 ENDIF ;
9
10
11
12
13
14
15 ==>

```

Screen: 50

```

0 ( FORTH colon decompiler )
1
2 : STG
3 PWORD 22 EMIT .WORD @
4 DUP COUNT TYPE 22 EMIT
5 C@ .WORD @ + 1+ .WORD ! ;
6
7 : CKIT
8 DUP CFALIT @BRANCH =
9 OVER CFALIT BRANCH = OR
10 OVER CFALIT (LOOP) = OR
11 OVER CFALIT (+LOOP) = OR
12 OVER CFALIT (/LOOP) = OR
12
13 IF BRNCH
14 ELSE DUP CFALIT LIT =
15 IF 1WORD ==>

```

Screen: 53

```

0 ( FORTH colon decompiler )
1
2 : DCMPR ( PFA -- )
3 DUP NFA CR CR DUP ID.
4 C@ 40 AND
5 IF ." (IMMEDIATE)"
6 ENDIF
7 CR CR ?DOCOL .WORD !
8 BEGIN NXT1 NP CKIT CR AGAIN ;
9
10 : DECOMP
11 [COMPILE] ' DCMPR ;
12
13
14
15 BASE ! ;S

```

Screen: 54

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 55

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 56

```
0 ( fig editor: TEXT LINE )
1
2 BASE @ HEX
3
4 ( This editor is based on the )
5 ( example editor supplied in )
6 ( the "fig-FORTH Installation )
7 ( Manual." )
8
9 : TEXT
10 HERE C/L 1+ BLANKS WORD
11 HERE PAD C/L 1+ CMOVE ;
12
13 : LINE
14 DUP FFF0 AND 17 ?ERROR
15 SCR @ (LINE) DROP ; ==>
```

Screen: 57

```
0 ( fig editor: MARK )
1
2 : MARK
3 10 0
4 DO
5 I LINE UPDATE
6 DROP
7 LOOP ;
8
9
10
11
12
13
14
15 -->
```

Screen: 58

```
0 ( fig editor: WHERE )
1
2 VOCABULARY EDITOR IMMEDIATE
3
4 ( Note: the fig bug is fixed )
5 ( in WHERE below. )
6
7 : WHERE ( n n -- )
8 2DUP DUP B/SCR / DUP SCR !
9 ." Scr # " DECIMAL . SWAP
10 C/L /MOD C/L * ROT BLOCK +
11 CR C/L -TRAILING TYPE
12 CR HERE C@ - 2- 0 MAX SPACES
13 1 2FE C! 1C EMIT 0 2FE C!
14 [COMPILE] EDITOR QUIT ; ==>
15
```

Screen: 59

```
0 ( fig editor: #L's -MOVE )
1
2 EDITOR DEFINITIONS
3
4 : #LOCATE ( -- n n )
5 R# @ C/L /MOD ;
6
7 : #LEAD ( -- n n )
8 #LOCATE LINE SWAP ;
9
10 : #LAG ( -- n n )
11 #LEAD DUP >R + C/L R> - ;
12
13 : -MOVE ( n n -- )
14 LINE C/L CMOVE UPDATE ; -->
15
```

Screen: 60

```

0 ( fig editor: H E S )
1
2 : H ( n -- )
3 LINE PAD 1+ C/L
4 DUP PAD C! CMOVE ;
5
6 : E ( n -- )
7 LINE C/L BLANKS UPDATE ;
8
9 : S ( n -- )
10 DUP 1- 0E
11 DO
12 I LINE
13 I 1+ -MOVE
14 -1 +LOOP
15 E ; ==>

```

Screen: 63

```

0 ( fig editor: I TOP )
1
2 : I ( n -- )
3 DUP S R ;
4
5 : TOP ( -- )
6 0 R# ! ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 61

```

0 ( fig editor: D M )
1
2 : D ( n -- )
3 DUP H 0F DUP ROT
4 DO
5 I 1+ LINE
6 I -MOVE
7 LOOP
8 E ;
9
10 : M ( n -- )
11 R# +! CR SPACE
12 #LEAD TYPE 14 EMIT #LAG
13 TYPE #LOCATE . DROP ;
14
15 -->

```

Screen: 64

```

0 ( fig editor: CLEAR COPY )
1
2 : CLEAR ( n -- )
3 SCR ! 10 0
4 DO
5 FORTH I EDITOR E
6 LOOP ;
7
8 : COPY ( n n -- )
9 B/SCR * OFFSET @ + SWAP
10 B/SCR * B/SCR OVER + SWAP
11 DO
12 DUP FORTH I BLOCK
13 2- ! 1+ UPDATE
14 LOOP
15 DROP ; ==>

```

Screen: 62

```

0 ( fig editor: T L R P )
1
2 : T ( n -- )
3 DUP C/L * R# !
4 DUP H 0 M ;
5
6 : L ( -- )
7 SCR @ LIST 0 M ;
8
9 : R ( n -- )
10 PAD 1+ SWAP -MOVE ;
11
12 : P ( n -- )
13 1 TEXT R ;
14
15 ==>

```

Screen: 65

```

0 ( fig editor: 1LINE FIND )
1
2 : 1LINE ( -- f )
3 #LAG PAD COUNT
4 MATCH R# +! ;
5
6 : FIND ( -- )
7 BEGIN
8 3FF R# @ <
9 IF
10 TOP PAD HERE C/L
11 1+ CMOVE 0 ERROR
12 ENDIF
13 1LINE
14 UNTIL ;
15 -->

```


Screen: 66

```
0 ( fig editor: DELETE )
1
2 : DELETE ( n -- )
3 >R #LAG + FORTH R -
4 #LAG R MINUS R# +! #LEAD
5 + SWAP CMOVE R) BLANKS
6 UPDATE ;
7
8
9
10
11
12
13
14
15 ==>
```

Screen: 69

```
0 ( End of fig-FORTH editor )
1
2 FORTH DEFINITIONS DCX
3
4
5
6
7
8
9
10
11
12
13
14
15 BASE !
```

Screen: 67

```
0 ( fig editor: N F B X )
1
2 : N ( -- )
3 FIND 0 M ;
4
5 : F ( -- )
6 1 TEXT N ;
7
8 : B ( -- )
9 PAD C@ MINUS M ;
10
11 : X ( -- )
12 1 TEXT FIND
13 PAD C@ DELETE
14 0 M ;
15 -->
```

Screen: 70

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 68

```
0 ( fig editor: TILL C )
1
2 : TILL ( -- )
3 #LEAD + 1 TEXT
4 1LINE 0= 0 ?ERROR
5 #LEAD + SWAP -
6 DELETE 0 M ;
7
8 : C
9 1 TEXT PAD COUNT #LAG ROT
10 OVER MIN >R FORTH R R# +!
11 R - >R DUP HERE R CMOVE
12 HERE #LEAD + R) CMOVE R)
13 CMOVE UPDATE 0 M ;
14
15 ==>
```

Screen: 71

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 72

```

0 ( Disk copy routines )
1 BASE @ DCX
2
3 0 VARIABLE SEC/PAS
4 0 VARIABLE SECNT
5
6 : AXLN ( system )
7 4 PICK 0
8 DO 3 PICK I 128 * +
9 3 PICK I + 3 PICK R/W
10 LOOP 2DROP 2DROP ;
11
12 : DCSTP
13 741 @ PAD DUP 1 AND - -
14 0 128 U/ SWAP DROP
15 SEC/PAS ! 0 SECNT ! ; ==>

```

Screen: 75

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 73

```

0 ( Disk copy routines )
1
2 : DISKCOPY1 ( -- )
3 DCSTP
4 BEGIN
5 CR CR ." Insert source and pu
6 sh START" WAIT
7 720 SECNT @ - SEC/PAS @ MIN
8 DUP >R PAD DUP 1 AND - SECNT
9 @ 2DUP 5 PICK (ROT 1 AXLN
10 CR CR ." Insert dest. and pu
11 sh START" WAIT 0 AXLN CR
12 R) SECNT +! SECNT @ DUP
13 ." sectors copied" 720 =
14 UNTIL EMPTY-BUFFERS
15 CR ." Done" CR ; -->

```

Screen: 76

```

0 ( 6502 Assembler in FORTH )
1 (
2 ( Originally written by
3 ( Patrick Mullarky.
4 (
5 ( Modified extensively 2/82
6 ( by Stephen Maguire,
7 ( Valpar International
8 (
9 (
10 ( This assembler conforms to the
11 ( fig "INSTALLATION GUIDE" and
12 ( to APX versions of FORTH.
13 (
14 ( )
15 ==>

```

Screen: 74

```

0 ( Disk copy routines )
1
2 : DISKCOPY2 ( -- )
3 DCSTP
4 CR ." Insert source in drive 1
5 " CR ." Insert dest. in drive 2
6 " CR ." Press START to copy"
7 WAIT
8 BEGIN
9 720 SECNT @ - SEC/PAS @ MIN
10 DUP >R PAD DUP 1 AND - SECNT
11 @ 2DUP 5 PICK (ROT
12 1 AXLN 720 + 0 AXLN
13 R) SECNT +! SECNT @ 720 =
14 UNTIL EMPTY-BUFFERS
15 CR ." Done" CR ; BASE !

```

Screen: 77

```

0 ( 6502 Assembler in FORTH )
1 (
2 ( Now supports:
3 (
4 ( IF,...ELSE,...ENDIF,
5 ( BEGIN,...WHILE,...REPEAT,
6 ( BEGIN,...AGAIN,
7 ( BEGIN,...any flag UNTIL,
8 ( C; & END-CODE
9 ( ;CODE
10 (
11 ( Also supports:
12 (
13 ( compiler security
14 ( definition checking )
15 -->

```

Screen: 78

```

0 ( 6502 Assembler in FORTH )
1 '( TRANSIENT TRANSIENT )( )
2 BASE @ HEX
3 ASSEMBLER DEFINITIONS
4
5 : SB
6   (BUILDS C, DOES) @ C, ;
7
8   000 SB BRK,      018 SB CLC,
9   008 SB CLD,      058 SB CLI,
10  088 SB CLV,      0CA SB DEX,
11  088 SB DEY,      0E8 SB INX,
12  0C8 SB INY,      0EA SB NOP,
13  048 SB PHA,      008 SB PHP,
14  068 SB PLA,      028 SB PLP,
15  040 SB RTI,      060 SB RTS, ==>

```

Screen: 79

```

0 ( 6502 Assembler in FORTH )
1 038 SB SEC,      0F8 SB SED,
2 078 SB SEI,      0AA SB TAX,
3 0BA SB TSX,      08A SB TXA,
4 09A SB TXS,      098 SB TYA,
5 0A8 SB TAY,
6
7 0 VARIABLE )J : ) 1 )J ! ;
8
9 : 3BY
10 (BUILDS C, DOES) C@ DUP 4C =
11 IF )J @ IF DROP 6C ENDIF
12 ENDIF C, , 0 )J ! ;
13
14 04C 3BY JMP,      06C 3BY JMP(),
15 020 3BY JSR,      06C 3BY )JMP, -->

```

Screen: 80

```

0 ( 6502 Assembler in FORTH )
1
2 : 256( DUP 100 ( HEX ) U( ;
3
4 70 CONSTANT VC ( over clear )
5 50 CONSTANT VS ( over set )
6 80 CONSTANT CC ( carry clear )
7 90 CONSTANT CS ( carry set )
8 D0 CONSTANT EQ ( zero )
9 F0 CONSTANT NE ( non-zero )
10 30 CONSTANT PL ( positive )
11 1 ?PAIRS 4C C, , ; IMMEDIATE
12
13 : IF,
14 C, 0 C, HERE 2 ; IMMEDIATE
15 ==>

```

Screen: 81

```

0 ( 6502 Assembler in FORTH )
1
2 : ENDIF,
3   DUP 2 = IF
4     DROP DUP HERE SWAP -
5     DUP 7F ) 5 ?ERROR
6     DUP -80 ( 5 ?ERROR
7     SWAP 1- C!
8   ELSE
9     3 ?PAIRS HERE SWAP !
10  ENDIF ; IMMEDIATE
11
12 : ELSE,
13   DUP 2 ?PAIRS 4C C, HERE 0 ,
14   (ROT [COMPILE] ENDIF, 3 ;
15   -->

```

Screen: 82

```

0 ( 6502 Assembler in FORTH )
1
2 : THEN,
3   [COMPILE] ENDIF, ; IMMEDIATE
4
5 : BEGIN,
6   HERE 1 ; IMMEDIATE
7
8 : UNTIL,
9   SWAP 1 ?PAIRS C,
10  HERE 1+ - DUP -80
11  ( 5 ?ERROR C, ; IMMEDIATE
12
13 : END,
14  [COMPILE] UNTIL, ; IMMEDIATE
15  ==>

```

Screen: 83

```

0 ( 6502 Assembler in FORTH )
1
2 : WHILE,
3   SWAP 1 ?PAIRS [COMPILE] IF,
4   DROP 4 ; IMMEDIATE
5
6 : REPEAT,
7   4 ?PAIRS SWAP 4C C, , 2
8   [COMPILE] ENDIF, ; IMMEDIATE
9
11 10 CONSTANT MI ( negative )
12
13 : END-CODE
14 [COMPILE] C; ; IMMEDIATE
15 -->

```

Screen: 84

```

0 ( 6502 Assembler in FORTH )
1 0D VARIABLE MODE ( ABS mode )
2 00 VARIABLE ACC ( A-reg? )
3
4 : BIT,
5 256< IF 24 C, C,
6 ELSE 2C C, , ENDIF ;
7
8 : CKMODE
9 MODE @ =
10 IF ( MODE = MODE - 8 )
11 256< ( if addr ( 256 )
12 IF
13 -08 MODE +!
14 ENDIF
15 ENDIF ; ==>

```

Screen: 85

```

0 ( 6502 Assembler in FORTH )
1
2 : M0
3 <BUILDS
4 C,
5 DOES>
6 SWAP 0D CKMODE
7 1D CKMODE SWAP
8 C@ MODE @ OR C,
9 256< IF C, ELSE , ENDIF
10 0D MODE ! ; ( ABS mode )
11
12 00 M0 ORA, 20 M0 AND,
13 40 M0 EOR, 60 M0 ADC,
14 80 M0 STA, A0 M0 LDA,
15 C0 M0 CMP, E0 M0 SBC, -->

```

Screen: 86

```

0 ( 6502 Assembler in FORTH )
1 : !ADDR C, 256< IF C, ELSE ,
2 ENDIF 0D MODE ! ;
3
4 : ZPAGE
5 OVER 100 U< IF F7 AND ENDIF ;
6
7 : M1
8 <BUILDS C, DOES> C@ ACC @
9 IF FB AND C, ELSE MODE @ 1D =
10 IF 10 ELSE 0 ENDIF OR ZPAGE
11 !ADDR ENDIF 0 ACC ! ;
12
13 00E M1 ASL, 02E M1 ROL,
14 04E M1 LSR, 06E M1 ROR,
15 0CE M1 DEC, 0EE M1 INC, ==>

```

Screen: 87

```

0 ( 6502 Assembler in FORTH )
1
2 : OPCODE
3 C@ ZPAGE MODE @ 1D =
4 MODE @ 19 = OR
5 IF 10 OR ENDIF ;
6
7 : M2
8 <BUILDS C,
9 DOES> OPCODE MODE @ 9 =
10 IF 4 - ENDIF !ADDR ;
11
12 : M3
13 <BUILDS C,
14 DOES> OPCODE !ADDR ;
15 -->

```

Screen: 88

```

0 ( 6502 Assembler in FORTH )
1 0AC M2 LDY, 0AE M2 LDX,
2 0CC M2 CPY, 0EC M2 CPX,
3 08C M3 STY, 08E M3 STX,
4
5 : X) 01 MODE ! ; ( [addr,X] )
6 : # 09 MODE ! ; ( immediate )
7 : )Y 11 MODE ! ; ( [addr],Y )
8 : ,X 1D MODE ! ; ( addr,X )
9 : ,Y 19 MODE ! ; ( addr,Y )
10 : .A 01 ACC ! ; ( a - reg )
11
12 0A SB ASL.A, 2A SB ROL.A,
13 4A SB LSR.A, 6A SB ROR.A,
14
15 ==>

```

Screen: 89

```

0 ( 6502 Assembler in FORTH )
1
2 : IFVC, VC [COMPILE] IF, ;
3 : IFVS, VS [COMPILE] IF, ;
4 : IFCC, CC [COMPILE] IF, ;
5 : IFCS, CS [COMPILE] IF, ;
6 : IFEQ, EQ [COMPILE] IF, ;
7 : IFNE, NE [COMPILE] IF, ;
8 : IFPL, PL [COMPILE] IF, ;
9 : IFMI, MI [COMPILE] IF, ;
10
11 : 0= EQ ; : 0< MI ; : >= EQ ;
12 : NOT 20 XOR ; : RP) 101 ,X ;
13 : BOT 0 ,X ; : SEC 2 ,X ;
14
15 -->

```

Screen: 90

```

0 ( End of 6502 assembler )
1 HEX
2 : XS,      XSAVE STX, ;
3 : XL,      XSAVE LDX, ;
4 : NXT,     NEXT JMP, ;
5 : POP,     POP JMP, ;
6 : POP2,    POPTWO JMP, ;
7 : PSH,     PUSH JMP, ;
8 : PSHA,    PUSH0A JMP, ;
9 : PUT,     PUT JMP, ;
10 : PUTA,   PUT0A JMP, ;
11
12
13 FORTH DEFINITIONS
14 '( PERMANENT PERMANENT )( )
15                               BASE !

```

Screen: 93

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 91

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 94

```

0 ( Buffer relocation )
1 BASE @ DCX
2
3 : RELOCBUFS      ( addr -- )
4   DUP 1 AND
5   IF CR ." Odd buffer address."
6     CR ." Try again." DROP QUIT
7   ENDIF
8   DUP           ' FIRST !
9   DUP 2112 + ' LIMIT !
10  DUP           PREV !
11  DUP           USE !
12  MTB CR 156 EMIT 156 EMIT
13  ." Buffers relocated to "
14  U.  ." and emptied" CR ;
15                                     ==>

```

Screen: 92

```

0 ( FORMAT )
1 BASE @ HEX
2
3 : FORMAT
4   CR CR ." Enter Drive#: " KEY
5   DUP EMIT 30 - 1 MAX 4 MIN CR
6   ." Hit RETURN to format drive "
7   DUP . CR
8   ." Hit any other key to abort "
9   KEY 9B =
10  IF (FMT) 1 = CR CR ." Format "
11    IF ." OK" ELSE ." ERROR"
12    ENDIF
13  ELSE CR ." Format aborted..."
14    DROP
15  ENDIF CR CR ;      BASE !

```

Screen: 95

```

0 ( Buffer relocation )
1
2 CR CR ." The buffers take 2112 b
3 ytes decimal." CR
4 CR ." To relocate buffers, put t
5 he new addr on stack and do:" CR
6 CR 7 SPACES ." RELOCBUFS FORGET
7 RELOCBUFS" CR CR  BASE !
8
9
10
11
12
13
14
15

```

Screen: 96

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 99

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 97

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 100

0 (Colors: hue CONSTANTS)
1
2 BASE @ DCX
3
4 0 CONSTANT GREY
5 1 CONSTANT GOLD
6 2 CONSTANT ORNG
7 3 CONSTANT RDORNG
8 4 CONSTANT PINK
9 5 CONSTANT LVNDR
10 6 CONSTANT BLPRPL
11 7 CONSTANT PRPLBL
12 8 CONSTANT BLUE
13 9 CONSTANT LTBLUE
14 10 CONSTANT TURQ
15 11 CONSTANT GRNBL ==>

Screen: 98

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 101

0 (Colors: hue CONSTANTS)
1
2 12 CONSTANT GREEN
3 13 CONSTANT YLWGRN
4 14 CONSTANT ORNGRN
5 15 CONSTANT LTORNG
6
7
8
9
10
11
12
13
14
15

BASE ! -->

Screen: 102

```

0 ( Colors: SETCOLOR BOOTCOLOR )
1 BASE @ DCX
2
3 : SETCOLOR      ( # hue lum -- )
4   SWAP 16 * OR SWAP
5   708 ( COLPF0 ) + C! ;
6
7 : SE.  SETCOLOR ;
8
9 : BOOTCOLOR      ( hue lum -- )
10  SWAP 16 * DUP 4 + DUP
11  [ ' COLD 35 + ] LITERAL C!
12  710 C! OR DUP
13  [ ' COLD 40 + ] LITERAL C!
14  709 C! ;
15                                BASE !

```

Screen: 105

```

0 ( Graphics: COLOR POS. LOC. )
1
2 0 VARIABLE CLRBYT
3
4 : COLOR          ( b -- )
5   CLRBYT ! ;
6
7 : POS.           ( h v -- )
8   54 C! 55 ! ;
9
10 : POSITION POS. ; ( h v -- )
11
12 : LOC.           ( x y -- b )
13   POS. CGET ;
14
15                                -->

```

Screen: 103

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 106

```

0 ( Graphics: CPUT )
1
2 HEX
3 CODE CPUT          ( b -- )
4   B5 C, 00 C, 48 C,
5   86 C, XSAVE C,
6   A2 C, 30 C, A9 C, 0B C,
7   9D C, 342 , 98 C,
8   9D C, 348 , 9D C, 349 ,
9   68 C, 20 C, C10 ,
10  A6 C, XSAVE C,
11  4C C, POP ,
12 C;
13
14
15                                ==>

```

Screen: 104

```

0 ( Graphics: CGET )
1
2 BASE @ DCX ' ( >SCD ) ( 68 KLOAD )
3
4 HEX
5 CODE CGET          ( -- b )
6   B5 C, 00 C, 48 C,
7   86 C, XSAVE C,
8   A2 C, 30 C, A9 C, 07 C,
9   9D C, 342 , 98 C,
10  9D C, 348 , 9D C, 349 ,
11  68 C, 20 C, C10 ,
12  A6 C, XSAVE C,
13  4C C, PUSH0A ,
14 C;
15                                ==>

```

Screen: 107

```

0 ( Graphics: POS@ POSIT PLOT )
1
2 : POS@           ( -- h v )
3   55 @ 54 C@ ;
4
5 : POSIT          ( h v -- )
6   POS. 54 C@ 5A C!
7   55 @ 5B ! ;
8
9 : PLOT           ( b h v -- )
10  POS. CLRBYT C@ CPUT ;
11
12
13
14
15                                -->

```

Screen: 108

```

0 ( Graphics:  GTYPE          )
1
2 : GTYPE          ( cnt adr -- )
3   0 MAX -DUP
4   IF 0+S
5     DO I C@ >SCD CLRBYT C@
6       40 * OR SCD> CPUT
7   LOOP
8   ENDIF ;
9
10
11
12
13
14
15                               ==>

```

Screen: 111

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 109

```

0 ( Graphics:  [G"]  G"          )
1
2 : (G")          ( -- )
3   R COUNT DUP 1+ R) + >R
4   GTYPE ;
5
6 : G"          ( -- )
7   22 STATE @
8   IF
9     COMPILE (G")
10    WORD HERE C@ 1+ ALLOT
11  ELSE
12    WORD HERE COUNT GTYPE
13  ENDIF ; IMMEDIATE
14
15                               -->

```

Screen: 112

```

0 ( Graphics Demo          )
1 BASE @ DCX
2
3 : BOX
4   1 COLOR 20 10 POSIT
5   50 10 DR.  50 28 DR.
6   20 28 DR.  20 10 DR. ;
7
8 : FBOX
9   5 GR. BOX
10  20 28 POS. 2 FIL ;
11
12
13 ( LOAD THIS SCREEN AND EXECUTE )
14 ( FBOX.  WHEN YOU'RE DONE, DO  )
15 ( FORGET BOX )          BASE !

```

Screen: 110

```

0 ( Graphics:  GCOM  DR.  FIL    )
1
2 CODE GCOM          ( n -- )
3   86 C, D1 C, B5 C, 00 C,
4   A2 C, 30 C, 9D C, 342 ,
5   20 C, C10 , A6 C, D1 C,
6   4C C, POP ,
7
8 : DR.          ( x y -- )
9   CLRBYT C@ 2FB C!
10  POS. 11 GCOM ;
11
12 : DRAWTO DR. ;
13
14 : FIL          ( fildat -- )
15  2FD C!  12 GCOM ;  BASE !

```

Screen: 113

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```


Screen: 114

```
0 ( Sound: SOUND SO. FILTER! )
1
2 BASE @ HEX
3 0 VARIABLE AUDCTL
4
5 : SOUND ( ch# freq dist vol --)
6 3 DUP D20F C! 232 C!
7 SWAP 10 * + ROT 2*
8 D200 + ROT OVER C! 1+ C!
9 AUDCTL C@ D208 C! ;
10
11 : SO. SOUND ;
12
13 : FILTER! ( b -- )
14 DUP D208 C! AUDCTL ! ;
15 ==>
```

Screen: 117

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 115

```
0 ( Sound: XSND XSND4 )
1
2 DCX
3
4 : XSND ( voice# -- )
5 2* 53761 +
6 0 SWAP C! ;
7
8 : XSND4 ( -- )
9 53760 8 0 FILL
10 0 FILTER! ;
11
12
13
14
15 BASE !
```

Screen: 118

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 116

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 119

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 120

```

0 ( Floating:  FDROP FDUP FSWAP )
1
2 BASE @  HEX
3
4 CODE FDROP          ( fp -- )
5   INX, INX, POPTWO JMP,
6 C;
7
8 CODE FDUP          ( fp -- fp fp )
9   # 6 LDY,
10  BEGIN,
11   DEX, 6 ,X LDA,  0 ,X STA,
12   DEY, 0=
13   UNTIL, NEXT JMP,
14 C;
15                               ==>

```

Screen: 121

```

0 ( Floating:  FSWAP FOVER          )
1
2 CODE FSWAP ( fp1 fp2 -- fp2 fp1 )
3   XSAVE STX,  # 6 LDY,
4   BEGIN,
5   0 ,X LDA,  PHA,  6 ,X LDA,
6   0 ,X STA,  PLA,  6 ,X STA,
7   INX, DEY, 0=
8   UNTIL, XSAVE LDX, NEXT JMP, C;
9
10 CODE FOVER ( fp fp -- fp fp fp )
11   # 6 LDY,
12   BEGIN,
13   DEX, 0C ,X LDA,  0 ,X STA,
14   DEY, 0=
15   UNTIL, NEXT JMP, C;      -->

```

Screen: 122

```

0 ( Floating:  conversion words )
1
2
3 CODE AFP
4   XS, D800 JSR, XL, NXT,
5 C;
6
7
8 CODE FASC
9   XS, D8E6 JSR, XL, NXT,
10 C;
11
12
13
14
15                               ==>

```

Screen: 123

```

0 ( Floating:  FADD FSUB FMUL ... )
1
2 CODE IFP  XS, D9AA JSR, XL, NXT,
3
4 CODE FPI   XS, D9D2 JSR, XL, NXT,
5
6 CODE FADD  XS, DA66 JSR, XL, NXT,
7
8 CODE FSUB  XS, DA60 JSR, XL, NXT,
9
10 CODE FMUL  XS, DADB JSR, XL, NXT,
11
12 CODE FDIV  XS, DB28 JSR, XL, NXT,
13
14 CODE FLG   XS, DECD JSR, XL, NXT,
15                               -->

```

Screen: 124

```

0 ( Floating:  FLG10 FEX FPOLY )
1
2 CODE FLG10
3   XS, DED1 JSR, XL, NXT,  C;
4
5 CODE FEX
6   XS, DDC0 JSR, XL, NXT,  C;
7
8 CODE FEX10
9   XS, DDCC JSR, XL, NXT,  C;
10
11 CODE FPOLY
12   XS, 0 ,X LDA,  PHA,
13   3 ,X LDA,  XSAVE LDY,
14   2 ,Y LDX,  TAY,  PLA,
15   DD40 JSR,  XL, POP2,  C;  ==>

```

Screen: 125

```

0 ( Floating:  system constants )
1
2 D4 CONSTANT FR0
3 E0 CONSTANT FR1
4 F3 CONSTANT INBUF
5 F2 CONSTANT CIX
6
7
8
9
10
11
12
13
14
15                               -->

```

Screen: 126

```

0 ( Floating: F@ F! F.TY      )
1
2 : F@                          ( a -- fp )
3   >R R @ R 2+
4   @ R> 4 + @ ;
5
6 : F!                          ( fp a -- )
7   >R R 4 + !
8   R 2+ ! R> ! ;
9
10 : F.TY                       ( a -- )
11   BEGIN
12     INBUF @ C@ DUP
13     7F AND EMIT
14     1 INBUF +! 80 )
15   UNTIL ;                      ==>

```

Screen: 127

```

0 ( Floating: F. F? <F >F      )
1
2 : F.                          ( fp -- )
3   FR@ F@ FSWAP FR@ F! FASC
4   F.TY SPACE FR@ F! ;
5
6 : F?                          ( a -- )
7   F@ F. ;
8
9 : <F                          ( fp fp -- )
10  FR1 F! FR@ F! ;
11
12 : >F                          ( -- fp )
13  FR@ F@ ;
14
15                                -->

```

Screen: 128

```

0 ( Floating: FS floating +--*/ )
1
2 : FS                          ( fp -- )
3   FR@ F! ;
4
5 : F+                          ( fp fp -- fp )
6   <F FADD >F ;
7
8 : F-                          ( fp fp -- fp )
9   <F FSUB >F ;
10
11 : F*                          ( fp fp -- fp )
12   <F FMUL >F ;
13
14 : F/                          ( fp fp -- fp )
15   <F FDIV >F ;                      ==>

```

Screen: 129

```

0 ( Floating: FLOAT FIX FLOG FEXP )
1
2 : FLOAT                      ( n -- fp )
3   FR@ ! IFP >F ;
4
5 : FIX                        ( fp -- n )
6   FS FPI FR@ @ ;
7
8 : LOG                        ( fp -- fp )
9   FS FLG >F ;
10
11 : LOG10                      ( fp -- fp )
12   FS FLG10 >F ;
13
14 : EXP                        ( fp -- fp )
15   FS FEX >F ;                      -->

```

Screen: 130

```

0 ( Floating: FEXP10 ASCF FLIT... )
1
2 : EXP10                      ( fp -- fp )
3   FS FEX10 >F ;
4
5 : ASCF                      ( a -- fp )
6   @ CIX ! INBUF ! AFP >F ;
7
8 : FLIT ( in dict. only: -- fp )
9   R> DUP 6 + >R F@ ;
10
11 : FLITERAL                  ( fp -- [fp] )
12   STATE @
13   IF
14     COMPILE FLIT HERE F! 6 ALLOT
15   ENDIF ; IMMEDIATE              ==>

```

Screen: 131

```

0 ( Floating: FLOATING FP      )
1
2 : FLOATING                  ( nnnn, -- fp )
3   BL WORD HERE 1+ ASCF
4   [COMPILE] FLITERAL ; IMMEDIATE
5
6 ( Float the following literal )
7 ( Ex: FLOATING 1.2345 )
8 ( or  FLOATING -1.67E-13 )
9
10 : FP                      ( nnnn, -- fp )
11   [COMPILE] FLOATING ;
12   IMMEDIATE
13
14
15                                -->

```

Screen: 132

```

0 ( Floating: FVARIABLE FCONSTANT)
1
2 : FVARIABLE      ( xxxx, fp -- )
3                  ( xxxx: -- a  )
4   <BUILDS
5     HERE F! 6 ALLOT
6   DOES> ;
7
8 : FCONSTANT      ( xxxx, fp -- )
9                  ( xxxx: -- fp )
10  <BUILDS
11    HERE F! 6 ALLOT
12  DOES> F@ ;
13
14
15                                ==>

```

Screen: 133

```

0 ( Floating: F@= F= F< F> )
1
2 : F@=            ( fp -- f )
3   OR OR @= ;
4
5 : F=             ( fp fp -- f )
6   F- F@= ;
7
8 : F<             ( fp fp -- f )
9   F- DROP DROP 80 AND 0) ;
10
11 : F>            ( fp fp -- f )
12   FSWAP F< ;
13
14
15                                BASE !

```

Screen: 134

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 135

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 136

```

0 ( Screen code conversion words )
1
2 BASE @ HEX
3
4 CODE >BSCD      ( a a n -- )
5   A9 C, 03 C, 20 C, SETUP ,
6   HERE C4 C, C2 C, D0 C, 07 C,
7   C6 C, C3 C, 10 C, 03 C, 4C C,
8   NEXT ,        B1 C, C6 C, 48 C,
9   29 C, 7F C, C9 C, 60 C, B0 C,
10  0D C, C9 C, 20 C, B0 C, 06 C,
11  18 C, 69 C, 40 C, 4C C, HERE
12  2 ALLOT 38 C, E9 C, 20 C, HERE
13  SWAP ! 91 C, C4 C, 68 C, 29 C,
14
15                                ==>

```

Screen: 137

```

0 ( Screen code conversion words )
1
2   80 C, 11 C, C4 C, 91 C, C4 C,
3   C8 C, D0 C, D3 C, E6 C, C7 C,
4   E6 C, C5 C, 4C C, , C;
5
6 CODE BSCD>      ( a a n -- )
7   A9 C, 03 C, 20 C, SETUP ,
8   HERE C4 C, C2 C, D0 C, 07 C,
9   C6 C, C3 C, 10 C, 03 C, 4C C,
10  NEXT ,        B1 C, C6 C, 48 C,
11  29 C, 7F C, C9 C, 60 C, B0 C,
12  0D C, C9 C, 40 C, B0 C, 06 C,
13  18 C, 69 C, 20 C, 4C C, HERE
14  2 ALLOT 38 C, E9 C, 40 C, HERE
15                                -->

```

Screen: 138

```

0 ( Screen code conversion words )
1
2 SWAP ! 91 C, C4 C, 68 C, 29 C,
3 80 C, 11 C, C4 C, 91 C, C4 C,
4 C8 C, D0 C, D3 C, E6 C, C7 C,
5 E6 C, C5 C, 4C C, ,
6
7
8 : >SCD SP@ DUP 1 >BSCD ;
9 : SCD> SP@ DUP 1 BSCD> ;
10
11
12
13
14
15 BASE !

```

Screen: 139

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 140

```

0 ( ValFORTH Video editor V1.0 )
1
2 BASE @ DCX ' ( >SCD ) ( 68 KLOAD )
3
4 VOCABULARY EDITOR IMMEDIATE
5 EDITOR DEFINITIONS
6
7 0 VARIABLE XLOC ( X coord. )
8 0 VARIABLE YLOC ( Y coord. )
9 0 VARIABLE LSTCHR ( last key )
10 0 VARIABLE ?ESC ( coded char? )
11 0 VARIABLE TBLK ( top block )
12 0 VARIABLE UPSTAT 2 ALLOT ( map )
13
14 15 CONSTANT 15 32 CONSTANT 32
15 128 CONSTANT 128 ==>

```

Screen: 141

```

0 ( ValFORTH Video editor V1.0 )
1
2 : LMOVE 32 CMOVE ;
3
4 : BOL 88 @ YLOC @ 1+ 32 * + ;
5
6 : SBL 88 @ 544 + ;
7
8 : CURLOC ( --- )
9 BOL XLOC @ + ; ( SCR ADDR )
10
11 : CSHOW ( --- )
12 CURLOC DUP ( GET SCR ADDR )
13 C@ 128 OR ( INVERSE CHAR )
14 SWAP C! ; ( STORE ON SCR )
15 -->

```

Screen: 142

```

0 ( ValFORTH Video editor V1.0 )
1
2 : CBLANK ( --- )
3 CURLOC DUP C@ 127
4 AND SWAP C! ;
5
6 : UPCUR ( -- )
7 CBLANK YLOC @ 1- DUP
8 0< IF DROP 15 ENDIF
9 YLOC ! CSHOW ;
10
11 : DNCUR ( -- )
12 CBLANK YLOC @
13 1 + DUP 15 )
14 IF DROP 0 ENDIF
15 YLOC ! CSHOW ; ==>

```

Screen: 143

```

0 ( ValFORTH Video editor V1.0 )
1
2 : LFCUR ( -- )
3 CBLANK XLOC @
4 1 - DUP 0< ( AT L-SIDE? )
5 IF DROP 31 ENDIF ( FIX IF SO )
6 XLOC ! CSHOW ;
7
8 : RTCUR ( -- )
9 CBLANK XLOC @
10 1+ DUP 31 ) ( AT R-SIDE? )
11 IF DROP 0 ENDIF ( FIX IF SO )
12 XLOC ! CSHOW ;
13
14 : EDMRK
15 1 YLOC @ 4 / UPSTAT + C! ; -->

```

Screen: 144

```

0 ( ValFORTH Video editor V1.0 )
1
2 : LNINS ( -- )
3 CBLANK
4 4 YLOC @ 4 /
5 DO 1 I UPSTAT + C! LOOP
6 YLOC @ 15 (
7 IF
8 BOL DUP 32 +
9 15 YLOC @ - 32 *
10 CMOVE
11 ENDIF
12 BOL 32 ERASE
13 CSHOW EDMRK ;
14
15 ==>

```

Screen: 145

```

0 ( ValFORTH Video editor V1.0 )
1
2 : LNDEL ( -- )
3 CBLANK
4 4 YLOC @ 4 /
5 DO 1 I UPSTAT + C! LOOP
6 YLOC @ 15 (
7 IF BOL ( FROM )
8 DUP 32 + SWAP ( TO )
9 15 YLOC @ - 32 * ( # CH )
10 CMOVE
11 ENDIF
12 BOL 15 YLOC @ -
13 32 * + 32 ERASE
14 CSHOW EDMRK ;
15 -->

```

Screen: 146

```

0 ( ValFORTH Video editor V1.0 )
1
2 : RUB ( -- )
3 XLOC @ 0 = NOT ( ON L-EDGE? )
4 IF LFCUR @ CURLOC C!
5 CSHOW EDMRK
6 ENDIF ;
7
8 : PTCHR ( -- )
9 EDMRK
10 LSTCHR @ 127 AND
11 DUP LSTCHR !
12 >SCD CURLOC C!
13 RTCUR XLOC @ 0 =
14 IF DNCUR ENDIF
15 0 ?ESC ! CSHOW ; ==>

```

Screen: 147

```

0 ( ValFORTH Video editor V1.0 )
1
2 : SCRSV ( -- )
3 88 @ 32 + PAD 512 BSCD)
4 4 0
5 DO
6 I UPSTAT + C@
7 0 I UPSTAT + C!
8 IF
9 PAD 128 I * +
10 TBLK @ I + BLOCK
11 128 CMOVE UPDATE
12 ENDIF
13 LOOP
14 0 XLOC ! 0 YLOC ! ;
15 -->

```

Screen: 148

```

0 ( ValFORTH Video editor V1.0 )
1
2 : SCRGT ( -- )
3 4 0
4 DO
5 TBLK @
6 I + BLOCK
7 PAD 128 I * +
8 128 CMOVE
9 LOOP
10 PAD 88 @ 32 +
11 512 >BSCD ;
12
13
14
15 ==>

```

Screen: 149

```

0 ( ValFORTH Video editor V1.0 )
1
2 : NWSCR ( -1/0/1 -- )
3 CBLANK DUP
4 IF SCRSV ENDIF 2* 2*
5 TBLK @ + 0 MAX TBLK ! SCRGT
6 TBLK @ 8 /MOD
7 DUP (ROT SCR !
8 IF 44 ELSE 53 ENDIF
9 ?1K NOT
10 IF
11 44 = SWAP 2* + DUP SCR ! 0
12 ENDIF
13 88 @ 17 + C!
14 0 84 C! 11 85 ! 1 752 C!
15 . 2 SPACES CSHOW ; -->

```

Screen: 150

```

0 ( ValFORTH Video editor  V1.0 )
1
2 : SPLCHR 1 ?ESC ! ;      ( -- )
3
4 : EXIT                      ( -- )
5   CBLANK 19 LSTCHR !
6   0 XLOC ! 0 YLOC ! ;
7
8 : EDTABT                    ( -- )
9   UPSTAT 4 0 FILL
10  EXIT ;
11
12
13
14
15                               ==>

```

Screen: 153

```

0 ( ValFORTH Video editor  V1.0 )
1
2 : (V)                      ( TBLK -- )
3   DECIMAL
4   DUP BLOCK DROP TBLK !
5   UPSTAT 4 0 FILL
6   1 PFLAG ! 0 GR.
7   1 752 C! CLS
8   1 559 C@ 252
9   AND OR 559 C!
10  112 560 @ 6 + C!
11  112 560 @ 23 + C!
12  ." Screen #" 11 SPACES
13  ." ValFORTH"
14  0 NWSCR
15                               -->

```

Screen: 151

```

0 ( ValFORTH Video editor  V1.0 )
1
2 : CONTROL                  ( n -- )
3   DUP 19 = IF DROP EXIT ELSE
4   DUP 17 = IF DROP EDTABT ELSE
5   DUP 28 = IF DROP UPCUR ELSE
6   DUP 29 = IF DROP DNCUR ELSE
7   DUP 30 = IF DROP LFCUR ELSE
8   DUP 31 = IF DROP RTCUR ELSE
9   DUP 126 = IF DROP RUB ELSE
10  DUP 157 = IF DROP LNINS ELSE
11  DUP 156 = IF DROP LNDEL ELSE
12    27 = IF DROP SPLCHR ELSE
13
14
15                               -->

```

Screen: 154

```

0 ( ValFORTH Video editor  V1.0 )
1
2
3 ( Main loop of editor )
4
5   BEGIN
6     KEY DUP LSTCHR !
7     ?ESC @
8     IF
9       PTCHR 0 LSTCHR !
10    ELSE
11      CONTROL
12      ENDIF
13      LSTCHR @ 19 =
14    UNTIL
15                               ==>

```

Screen: 152

```

0 ( ValFORTH Video editor  V1.0 )
1
2   PTCHR ( IF NOTHING SPECIAL )
3   ENDIF ENDIF ENDIF ENDIF
4   ENDIF ENDIF ENDIF ENDIF
5   ENDIF ENDIF ;
6
7
8
9
10
11
12
13
14
15                               ==>

```

Screen: 155

```

0 ( ValFORTH Video editor  V1.0 )
1
2   CBLANK SCR SV 0 767 C!
3   2 560 @ 6 + C!
4   2 560 @ 23 + C!
5   2 559 C@ 252
6   AND OR 559 C!
7   0 752 C! CLS CR
8   ." Last edit on screen # "
9   SCR @ . CR CR ;
10
11  FORTH DEFINITIONS
12
13 : V                      ( s -- )
14   1 MAX B/SCR *
15  EDITOR (V) ;
                               -->

```

Screen: 156

```
0 ( ValFORTH Video editor V1.0 )
1
2 : L ( -- )
3 SCR @ DUP 1+
4 B/SCR * SWAP B/SCR *
5 EDITOR TBLK @ DUP (ROT
6 (= (ROT ) AND
7 IF
8 EDITOR TBLK @
9 ELSE
10 SCR @ B/SCR *
11 ENDIF
12 EDITOR (V) ;
13
14
15 ==>
```

Screen: 159

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 157

```
0 ( ValFORTH Video editor V1.0 )
1
2 : CLEAR ( s -- )
3 B/SCR * B/SCR 0+S
4 DO
5 FORTH I BLOCK
6 B/BUF BLANKS UPDATE
7 LOOP ;
8
9 : COPY ( s1 s2 -- )
10 B/SCR * OFFSET @ +
11 SWAP B/SCR * B/SCR 0+S
12 DO DUP FORTH I
13 BLOCK 2- !
14 1+ UPDATE
15 LOOP DROP ; -->
```

Screen: 160

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 158

```
0 ( ValFORTH Video editor V1.0 )
1
2
3 ( Note: the fig bug is fixed )
4 ( in WHERE below. )
5
6 HEX
7 : WHERE ( [ n n ] -- )
8 2DUP DUP B/SCR / DUP SCR !
9 ." Scr # " DECIMAL . SWAP
10 C/L /MOD C/L * ROT BLOCK +
11 CR C/L -TRAILING TYPE
12 CR HERE C@ - 2- 0 MAX SPACES
13 1 2FE C! 1C EMIT 0 2FE C!
14 [COMPILE] EDITOR QUIT ;
15 BASE !
```

Screen: 161

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```


Screen: 162

```

0 ( DOS:  input/output routines )
1
2 BASE @ HEX
3
4 340 VARIABLE IOCB
5 0 VARIABLE IO.X
6 0 VARIABLE IO.CH
7
8 : IOCC
9 10 * 70 MIN DUP IO.X C!
10 340 + IOCB ! ;
11
12 : <IO>
13 <BUILDS ,
14 DOES> @ IOCB @ + ;
15

```

==>

Screen: 165

```

0 ( DOS:  GET/PUTREC STATUS DEV )
1
2 : GETREC      ( adr n1 n2 -- n3 )
3 IOCC 5 ICCOM C! ICBLL !
4 ICBAL ! XCIO ;
5
6 : PUTREC      ( adr n1 n2 -- n3 )
7 IOCC 9 ICCOM C! ICBLL !
8 ICBAL ! XCIO ;
9
10 : STATUS      ( n1 -- n2 )
11 IOCC ICSTA C@ ;
12
13 : DEVSTAT      ( n1 -- n2 n3 n4 )
14 IOCC 0D ICCOM C! XCIO
15 >R 2EA @ 2EC @ R) ;

```

-->

Screen: 163

```

0 ( DOS:  system words )
1
2 2 <IO> ICCOM 3 <IO> ICSTA
3 4 <IO> ICBAL 8 <IO> ICBLL
4 A <IO> ICAX1 B <IO> ICAX2
5 C <IO> ICAX3 D <IO> ICAX4
6 E <IO> ICAX5 F <IO> ICAX6
7
8
9 CODE XCIO
10 XSAVE STX, IO.X LDX,
11 IO.CH LDA, E456 JSR,
12 XSAVE LDX, IO.CH STA,
13 TYA, PUSH0A JMP,
14 C;
15

```

-->

Screen: 166

```

0 ( DOS:  SPECIAL )
1
2 : SPECIAL
3 ( n1 n2 n3 n4 n5 n6 n7 n8 -- n9)
4 IOCC ICCOM C! ICAX6 C!
5 ICAX5 C! ICAX4 C! ICAX3 C!
6 ICAX2 C! ICAX1 C! XCIO ;
7
8
9
10
11
12
13
14
15

```

BASE !

Screen: 164

```

0 ( DOS:  OPEN CLOSE PUTC GETC )
1
2 : OPEN      ( adr n1 n2 n3 -- n4 )
3 IOCC ICAX2 C! ICAX1 C!
4 ICBAL ! 03 ICCOM C! XCIO ;
5
6 : CLOSE      ( n1 -- )
7 IOCC 0C ICCOM C! XCIO DROP ;
8
9 : PUT      ( c n1 -- n2 )
10 IOCC IO.CH C! 0B
11 ICCOM C! XCIO ;
12
13 : GET      ( n1 -- c n2 )
14 IOCC 7 ICCOM C! XCIO
15 IO.CH C@ SWAP ;

```

==>

Screen: 167

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 168

```

0 ( Atari 850 download )
1
2 BASE @ HEX
3
4 CODE DO-SIO
5 XSAVE STX, 0 # LDA,
6 E459 JSR,
7 XSAVE LDX, NEXT JMP,
8
9 : SET-DCB
10 50 300 C! 1 301 C!
11 3F 302 C! 40 303 C!
12 500 304 ! 5 306 C!
13 0 307 C! C 308 C!
14 0 309 ! 0 30B C! ;
15 ==>

```

Screen: 171

```

0 CONTENTS OF THIS DISK, cont:
1
2 fig EDITOR: 56 LOAD
3 BUFFER RELOCATION: 94 LOAD
4 AUTO-BOOT UTILITY: 30 LOAD
5 OPERATING SYS. WORDS: 162 LOAD
6 850 DOWNLOAD (RS-232): 168 LOAD
7 (OPSYS AND 850 NEED ASSEMBLER)
8
9
10
11
12
13
14
15

```

Screen: 169

```

0 ( Atari 850 download )
1
2 CODE RELOCATE
3 XSAVE STX, 506 JSR,
4 HERE 8 + JSR, XSAVE LDX,
5 NEXT JMP, 0C )JMP,
6
7
8 : RS232 ( -- )
9 HERE 2E7 ! SET-DCB DO-SIO
10 500 300 0C CMOVE DO-SIO
11 RELOCATE 2E7 @ HERE - ALLOT
12 HERE FENCE ! ;
13
14
15 BASE !

```

Screen: 172

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 170

```

0 CONTENTS OF THIS DISK:
1
2 PRINTER UTILITIES: 38 LOAD
3 DEBUGGING AIDS: 42 LOAD
4 VALFORTH EDITOR 1.0: 140 LOAD
5 ASSEMBLER: 76 LOAD
6 COLOR COMMANDS: 100 LOAD
7 GRAPHICS: 104 LOAD
8 GRAPHICS DEMO: 112 LOAD
9 SOUNDS: 114 LOAD
10 FLOATING POINT: 120 LOAD
11 (FP REQUIRES ASSEMBLER FIRST)
12 SCREEN CODE CONVERS.: 136 LOAD
13 FORMATTER: 92 LOAD
14 DISK COPIERS: 72 LOAD
15 (continued on next screen)

```

Screen: 173

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages)
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 179

0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

GENERAL UTILITIES AND VIDEO EDITOR

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

Software and Documentation
© Copyright 1982
Valpar International

vaIFORTH T.M.

Screen Oriented Video Editor

Version 1.1
March 1982

The FORTH language is a very powerful addition to the Atari home computer. Programs which are impossible to write in BASIC (usually because of limitations in speed and flexibility) can almost always be written in FORTH. Even when one has mastered the BASIC language, making corrections or additions to programs can be tedious. The video editor described here removes this problem from the FORTH environment. Similar to the MEMO PAD function in the Atari operating system, this editor makes it possible to insert and delete entire lines of code, insert and delete single characters, toggle between insert and replace modes, move entire blocks of text, and much more.

Software and Documentation
© Copyright 1982
Valpar International

valFORTH^{TM.}
SOFTWARE SYSTEM

GENERAL UTILITIES AND VIDEO EDITOR

Stephen Maguire
Evan Rosen

Software and Documentation
©Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

VALPAR INTERNATIONAL

Disclaimer of Warranty on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

valFORTH UTILITIES/EDITOR USER'S MANUAL

Table of Contents

	<u>Page</u>
XI. valFORTH 1.1 DISPLAY-ORIENTED VIDEO EDITOR A user's manual for the valFORTH video editor	
a) OVERVIEW	1
b) ENTERING THE EDIT MODE	2
c) CURSOR MOVEMENT	5
d) EDITING COMMANDS	6
e) STORAGE BUFFER MANAGEMENT	8
f) CHANGING SCREENS; SAVING; ABORTING	13
g) SPECIAL COMMANDS	14
h) SCREEN MANAGEMENT	16
i) EDITOR COMMAND SUMMARY	18
 XII. STRINGS, ARRAYS, CASE STATEMENTS, DOUBLE NUMBER EXTENSIONS	
a) STRING PACKAGE	1
b) ARRAYS, TABLES, VECTORS	5
c) CASE:, CASE, SEL, COND	8
d) DOUBLE NUMBER EXTENSIONS	14
 XIII. HI-RES TEXT, MISC. UTILITIES, TRANSIENTS	
a) HI-RESOLUTION (8 GR.) TEXT OUTPUT	1
b) MISCELLANEOUS UTILITIES	5
c) TRANSIENTS (DISPOSABLE ASSEMBLERS, ETC.)	9
 XIV. UTILITIES/EDITOR SUPPLIED SOURCE LISTING	

GENERAL UTILITIES and VIDEO EDITOR

Overview

This editor is a powerful extension to the valFORTH system designed specifically for the Atari 400/800 series of microcomputers. The main purpose for this editor is to give the FORTH programmer an easy method of text entry to screens for subsequent compilation. The editor has four basic modes of operation:

- 1) It allows entering of new text to a FORTH screen as though typing on a regular typewriter.
- 2) It allows quick, painless modification of any text with a powerful set of single stroke editing commands.
- 3) It pinpoints exactly where a compilation error has occurred and sets up the editor for immediate correction and recompilation.
- 4) Given the name of a precompiled word, it locates where the original text definition of the word is on disk, if the "LOCATOR" option had been selected when the word was compiled.

The set of single stroke editing commands is a superset of the functions found in the MEMO PAD function of the standard Atari operating system. In addition to cursor movement, single character insertion/deletion, and line insertion/deletion, the editor supports a clear-to-end-of-line function, a split command which separates a single line into two lines, and a useful insert submode usually found only in higher quality word processors.

In addition, there are provisions for scrolling both forwards and backwards through screens, and to save or "forget" any changes made. This is useful at times when text is mistakenly modified.

Also provided is a visible edit storage buffer which allows the user to move, replace, and insert up to 320 lines of text at a time. This feature alone allows the FORTH programmer to easily reorganize source code with the added benefit of knowing that re-typing mistakes are avoided. Usage has shown that once edit-buffer management is learned, significant typing and programming time can be saved.

For those times when not programming, the editor can double as a simple word processor for writing letters and filling other documentation needs. The best method for learning how to use this powerful editor is to enter the edit mode and try each of the following commands as they are encountered in the reading.

As stated above, there are four ways in which to enter the video editor. The following four commands explain each of the possibilities. Note that the symbol "<ret>" indicates that the "RETURN" key is to be typed.

V

view screen

(scr# ---)

To edit a screen for the first time, the "View" command is to be used. The video display will enter a 32 character wide mode and will be broken into three distinct sections. For example,

50 V <ret>

should give something like the display shown in fig. 1.

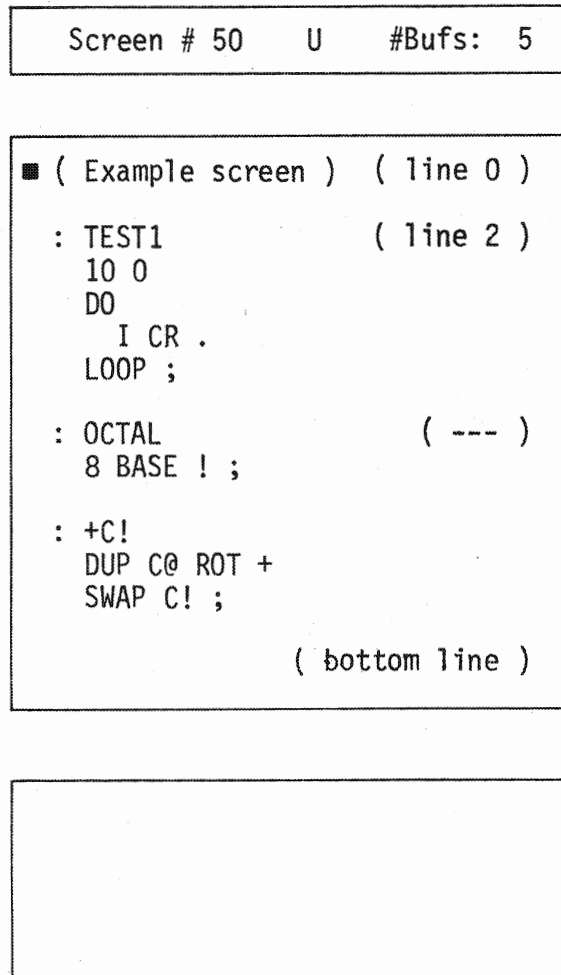


Fig. 1

The top window, composed of a single line, indicates in decimal which screen is currently being edited. One should always make a practice of checking this screen number to insure that editing will be done on the intended screen. Often times, when working with other number bases, the wrong screen is called up accidentally and catching this mistake early can save time. Also shown is the size of the edit buffer (described later). In this example, the buffer is five lines in length. This window is known as the heading window.

FORTH screens typically are 1K (1024 characters) long. Since it is impossible to see an entire screen simultaneously, this editor reveals only half a screen at a time. There is an "upper" half and a "lower" half. In the center of the heading window, either a "U" or an "L" is displayed indicating which half of the current screen is being viewed. If the valFORTH system is in the half-K screen mode, neither "U" nor "L" is displayed since an entire half-K screen can be viewed at one time. In figure 1, the upper half of a full-K screen is being viewed.

The second window (the text window) contains the text found on the specified screen. This window is 32 characters wide and 16 lines high. The white cursor (indicated by the symbol "■") will be in the upper-lefthand corner of the screen awaiting editing commands.

The final five-line window found at the bottom of the screen is known as the buffer window. This is used for advanced editing and is described in greater detail in the section entitled "Buffer Management."

L re-edit last screen (---)

This command is used to re-edit the "Last" screen edited. It functions identically to the "V" command described above, except no screen number is specified.

Example: L <ret> (re-edit screen 50)

WHERE find location of error (---)

If, when compiling code, a compilation error occurs, the WHERE command will enter the edit mode and position the cursor over the last letter of the offending word. The word can then be fixed and the screen can be re-compiled. Bear in mind that using the WHERE command prior to any occurrence of an error could give strange results.

LOCATE locate definition cccc (---)

Once source text has been compiled into the dictionary, it loses easy readability to all but experts of the FORTH language. Often times, though, it is helpful to see what the original source code was. The DECOMP command found in the debugger helps tremendously in this regard, however, some structures such as IF and DO are still difficult to follow. For this reason, the LOCATE command is included with the editor.

This command accepts a word name, and if at all possible it will actually direct the editor to load in the screen where that word was defined. This is very helpful at times when one cannot remember where the original text was. If the screen shown in figure 1 were loaded and the command

LOCATE +C! <ret>

were given, the editor would call up screen 50 and position the cursor over the word ":" which is the beginning of the definition for "+C!". Typically, the LOCATE command will point to ":", "CODE", "CONSTANT", and other defining words.

There is a drawback to this feature, however. In order to call up any word, the LOCATE command must know where the word actually is. Normally, when a word is compiled, there is no way of knowing where it was loaded from. Thus for the LOCATE command to work, each time a word is entered into the dictionary, three extra bytes of memory must be used to store this lookup information. For an application with many words, these extra bytes per word add up quickly, and this is not always desirable. For this reason, the LOCATOR command (described below) allows the user to enable or disable the storage of this lookup information. Only words that were compiled with the LOCATOR option selected can be located. If a word cannot be located, the user is warned, or if the DEBUGGER is loaded, the word is DECOMPed giving pseudo original code.

LOCATOR enable/disable location (ON/OFF ---)

In order for a word to be locatable using LOCATE, the LOCATOR option must have been selected prior to compiling the word. The LOCATOR option is selected by executing "ON LOCATOR" and deselected by executing "OFF LOCATOR". For example:

```
ON LOCATOR
: PLUS . " = " + . ;           (partial view of a screen)
: STAR 42 EMIT ;
OFF LOCATOR
: NEGATE MINUS ;
```

Only the words PLUS and STAR can be located. NEGATE cannot be located since the LOCATOR option was disabled. If the DEBUGGER were loaded, NEGATE would be decompiled (see the debugger), otherwise, the user would be given a warning. The default value for LOCATOR is OFF.

```
#BUFS      set buffer length      ( #lines --- )
```

The #BUFS command allows the user to specify the length (in terms of number of lines) of the special edit storage buffer. The power of the edit buffer lies in the number of lines that can be stored in it. Although the default value is five, practice shows that at least 16 lines should be set aside for this buffer. The maximum number of lines allowable is 320 which is enough to hold 20 full screens simultaneously.

The following sections give a detailed description of all commands which the video editor recognizes. A quick reference command list can be found following these descriptions.

Cursor Movement

When the edit mode is first entered via the "V" command, a cursor is placed in the upper lefthand corner of the screen. It should appear as a white block and may enclose a black letter. Whenever any key is typed and it is not recognized as an editor command, it is placed in the text window where the cursor appears. Likewise, any line functions (such as delete line) work on the line where the cursor is found.

`ctrl^`, `ctrl v`, `ctrl <`, `ctrl >` move-cursor commands

To change the current edit line or character, one of four commands may be given. These are known as cursor commands. They are the four keys with arrows on them. These keys move the cursor in the direction specified by the arrow on the particular key pressed. There are times, however, when this is not the case.

If the current cursor line is the topmost line of the text window, and the "cursor-up" command is issued (by simultaneously typing "ctrl" and "up-arrow"), the cursor will move to the bottom line of the text window. Likewise, a subsequent "cursor-down" command would return the cursor to the topmost line of the window. Similarly, if the cursor is positioned on the leftmost edge and the "cursor-left" command is given, the cursor will "wrap" to the rightmost character ON THE SAME LINE. Issuing "cursor-right" will wrap back to the first character on that line.

RETURN next-line command

Normally, the RETURN key positions the cursor on the first character of the next line. If RETURN is pressed when the cursor is on the last line of the text window (i.e., when the last text line of the screen is current), the cursor is positioned in the upper lefthand corner of the screen.

TAB tabulate command

The TAB key is used to tabulate to the next fixed four column tabular stop to the right of the current cursor character. TABbing off the end of the current line simply places the cursor at the beginning of that same line.

NOTE:

Many commands in the editor will "mark" a current FORTH screen as updated so that any changes made can be preserved on disk. As simple cursor movement does not change the text window in any way, these commands never mark the current FORTH screen. See the section on screen management for more information.

Editing Commands

Editing commands are those commands which modify the text in some predefined manner and mark the current FORTH screen as updated for later saving.

ctrl INS character insert command

When the "insert-character" command is given, a blank character is inserted at the current cursor location. The current character and all characters to the right are pushed to the right by one character position. The last character of the line "falls off" the end and is lost. The inserted blank then becomes the current cursor character. This is the logical complement to the "delete-character" command described below.

ctrl	DEL	delete character command
------	-----	--------------------------

When the "delete-character" command is issued, the current cursor character is removed, and all characters to the right of the current cursor character are moved left one position, thus giving a "squeeze" effect. This is normally called "closing" a line. The rightmost character on the line (which was vacated) is replaced with a blank. This serves as the logical complement to the "insert-command" described above.

shift INS line insert command

The "line-insert" command inserts a blank line between the current cursor line and the line immediately above it. The current line and all lines below it are moved down one line to make room for the new line. The last line on the screen falls off the bottom and is lost. If this command is accidentally typed, the "oops" command (ctrl-O) described later can be used to recover from the mistake. Also see the "from buffer" command described in the section on buffer management for a similar command. This command serves as the logical complement to the "line-delete" command described below.

shift DEL line delete command

The "line-delete" command deletes the current cursor line. All lines below the current line are brought up one line and a blank line fills the vacated bottom line of the text window. The deleted line is lost. If this command is accidentally issued, recovery can be made by issuing the "oops" command (ctrl-0) described later. Also see the "to-buffer" command described in the section on buffer management for a similar command. The "delete-line" command serves as the logical complement to the "line-insert" command.

ctrl H

erase to end of line

The "Hack" command performs a clear-to-end-of-line function. The current cursor character and all characters to the right of it on the current line are blank filled. All characters blanked are lost. The "oops" command described later can be used to recover from an accidentally hacked line.

ctrl I

insert/replace toggle

In normal operation, any key typed which is not recognized by the editor as a control command will replace the current cursor character with itself. This is the standard replace mode. Normally, if one wanted to insert a character at the current cursor location, the insert character command would have to be issued before any text could be entered. If inserting many characters, this is cumbersome.

When active, the insert submode automatically makes room for any new characters or words and frees the user from having to worry about this. When the editor is called up via the "V" command, the insert mode is deactivated. Issuing the insert toggle command will activate it and the cursor will blink, indicating that the insert mode is on. Issuing the command a second time will deactivate the insert mode and restore the editor to the replace mode. Note that while in the insert mode, all edit commands (except BACKS, below) function as before.

BACKS

delete previous character

The BACKS key behaves in two different ways, depending upon whether the editor is in the insert mode or in the replace mode. When issued while in the replace mode, the cursor is backed up one position and the new current character is replaced with a blank. If the cursor is at the beginning of the line, the cursor does not move, but the cursor character is still replaced with a blank.

If the editor is in the insert mode, the cursor backs up one position, then deletes the new current cursor character and then closes the line. If the cursor is at the beginning of the line, the cursor remains in the same position, the cursor character is deleted and the line closed.

NOTE:

As all of the above commands modify the text window in some manner, the screen is marked as having been changed. This is to be sure that all changes made are eventually saved on disk. The "quit" command described in the section on changing screens allows one to unmark a screen so that major mistakes need not be saved.

Buffer Management

Much of the utility of the valFORTH editor lies in its ability to temporarily save text in a visible buffer. To aid the user, it is possible to temporarily send text to the buffer and to later retrieve it. This storage buffer can hold as many as 320 lines of text simultaneously. This buffer is viewed through a 5 line "peephole" visible as the last window on the screen. Using this buffer, it is possible to duplicate, move, and easily reorganize text, in addition to temporarily saving a line that is about to be edited so that the original form can be viewed or restored if necessary. The following section will explain exactly how to accomplish each of these actions.

ctrl T to buffer command

The "to-buffer" command deletes the current cursor line, but unlike the "delete-line" command where the line is lost, this command moves the "peephole" down and copies the line to the bottom line of the visible buffer window. This line is the current buffer line. The buffer is rolled upon each occurrence of this command so that it may be used repeatedly without the loss of stored text.

For example, if the cursor is positioned on line eight of the display shown in figure 1 and the "to-buffer" command is issued twice, the final result will be as shown in figure 2.

ctrl F from buffer command

The "from-buffer" command does exactly the opposite of the "to-buffer" command described above. It takes the current buffer line and inserts it between the current cursor line and the line above it. The cursor line and all lines below it are moved down one line with the last line of the text window being lost. If the cursor were placed on line 14 of the above screen display and the "from-buffer" command were issued once, the display in figure 3 would result.

Screen # 50 U #Bufs: 5

Current:

```
( Example screen ) ( line 0 )

: TEST1                ( line 2 )
  10 0
  DO
    I CR .
  LOOP ;

■
: +C!
  DUP C@ ROT +
  SWAP C! ;

( bottom line )
```

Current:

```
: OCTAL                ( --- )
  8 BASE ! ;
```

fig. 2

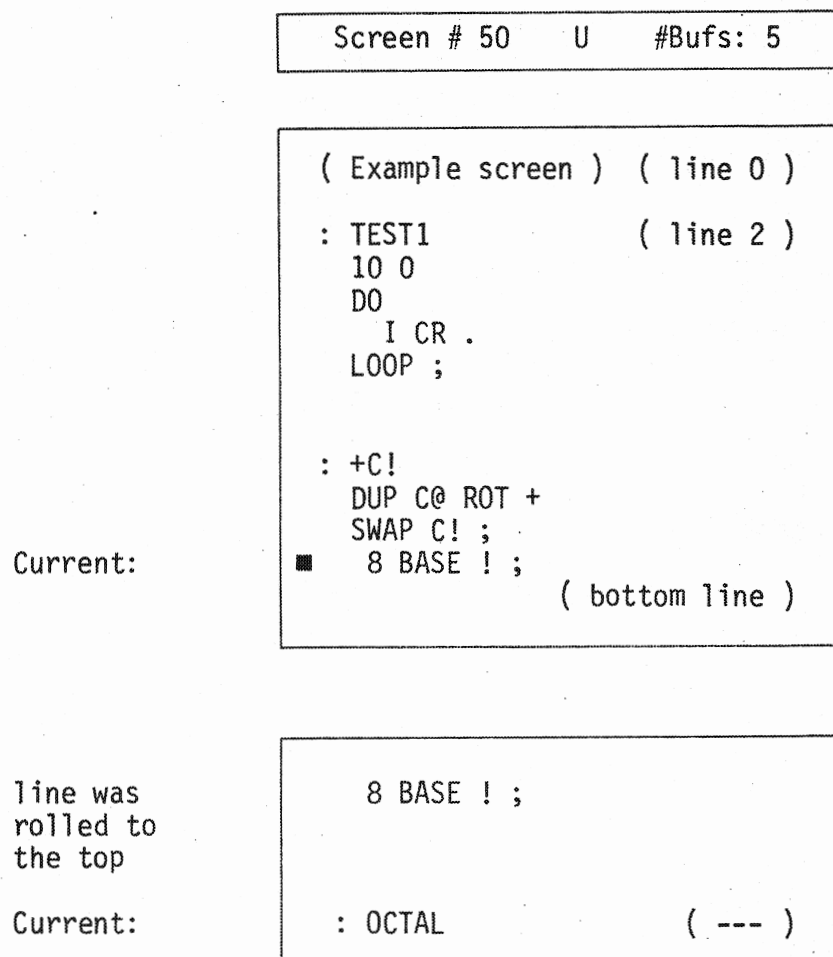


fig. 3

If the "from-buffer" command is issued again, then lines 13 through 15 of the text window would look like:

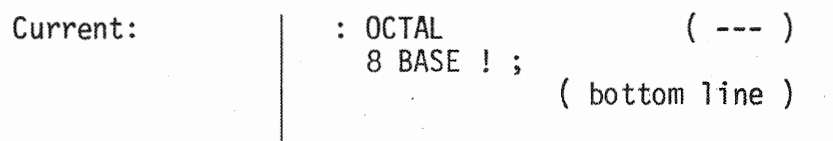


fig. 4

Note that a block of text has been moved on the screen. Larger blocks of text can be moved in the same manner.

ctrl K

copy to buffer command

The "copy-to-buffer" command takes the current cursor line and duplicates it, sending the copy to the buffer. This command functions identically to the "to-buffer" command described above, except that the current cursor line is NOT deleted from the text window.

ctrl U

copy from buffer

The "copy-from-buffer" command replaces the current cursor line with the current buffer line. This command functions identically to the "from-buffer" command described above, except that the buffer line is not inserted into the text window, it merely replaces the current cursor line. The "oops" command described below can be used to recover from accidental usage of this command.

ctrl R

roll buffer

The "roll-buffer" command moves the buffer "peephole" down one line and redisplay the visible window. If the buffer were the minimum five lines in length, the bottom four lines in the window would move up a line and the top line would "wrap" to the bottom and become the current buffer line. If there were more than five buffer lines, the bottom four lines would move up a line, the topmost line would be pushed up behind the peephole, and a new buffer line coming up from below the peephole would be displayed and made current. For example, if the buffer were five lines long and contained:

Current:

```
( Who?   )
( What?  )
( When?  )
( Where? )
( Why?   )
```

Fig. 5

the "roll-buffer" command gives:

Current:

```
( What?  )
( When?  )
( Where? )
( Why?   )
( Who?   )
```

Fig. 6

ctrl B back-roll-buffer command

The "back-roll-buffer" does exactly the opposite of the "roll-buffer" command described above. For example, if given the buffer in figure 6 above, the "back-roll" command would give the buffer shown in figure 5.

ctrl C clear buffer line command

The "clear-buffer-line" command clears the current buffer line and then "back-rolls" the buffer so that successive clears can be used to erase the entire buffer.

NOTE:

Any of the above commands which change the text window will mark the current screen as updated. Those commands which alter only the buffer window (such as the "roll" command) will not change the status of the current screen.

Changing Screens

There are four ways in which to leave a FORTH screen. These four methods are: moving to a previous screen, moving to a following screen, saving the current screen and exiting, or simply aborting the edit session. The four commands allowing this are now described:

```
ctrl P          previous screen command
```

The "previous-screen" command has two basic functions. If the lower part of the current screen is being viewed in the text window, this command simply displays the upper portion of the screen. If the upper portion is already being viewed, then the "previous-screen" command saves any changes made to the current screen and then loads in the screen immediately before the current screen. The lower part of the screen will then be displayed. If in the half-K screen mode, however, this command simply changes screens.

```
ctrl N          next screen command
```

Like the "previous-screen" command described above, the "next-screen" command also has two basic functions. If the upper part of a screen is being viewed, this command simply displays the lower portion. If, on the other hand, the lower part of the screen is being edited, any changes made to the current screen are saved and the next screen is loaded.

```
ctrl S          save command
```

The "save" command saves any changes made to the current screen and exits the edit mode. The video screen is cleared, and the number of the screen just being edited is displayed for reference. Note that it is usually a good idea to immediately FLUSH (described in the section on screen management below) any unsaved screens.

ctrl Q quit command

The "quit" command aborts the edit session "forgetting" any changes made to the text visible in the text window. Changes made on previously edited screens will NOT be forgotten. The "quit" command is usually used when either the wrong screen has been called up, or if it becomes desirable to start over and re-edit the screen again.

Special Commands

There are four special commands in this editor which allow greater flexibility in programming on the valFORTH system:

ESCAPE

special key command

The "special-key" command instructs the video editor to ignore any command function of the key typed next and force a character to the screen. For example, normally when "ctrl >" is typed, the cursor is moved right. By typing "ESCAPE ctrl >" the cursor is not moved -- rather, the right-arrow is displayed.

ctrl A

arrow command

When dealing with FORTH screens, it is often necessary to put the FORTH word "-->" (pronounced "next screen") or the ValFORTH word "==">" (pronounced "next screen") or the ValFORTH word "==">" (pronounced "next half-K screen") at the end of a screen for chaining a long set of words together. This command automatically places, or erases, an arrow in the lower right hand corner of the text window. If "-->" is already there, it is replaced with "==">". If "==">" is found, it is erased. (This command marks the screen as updated.)

ctrl J

split line command

Often times, for formatting reasons, it is necessary to "split" a line into two lines. The split line command takes all characters to the left of the cursor and creates the first line, and with the remaining characters of the original line, a second line is created. Graphically, this looks like:

before: | The quick■brown fox jumped.

```
after: | The quick
        |      brown fox jumped.
```

Since a line is inserted, the bottom line of the text window is lost. Using the "oops" command below, however, this can be recovered.

ctrl 0

oops command

Occasionally, a line is inserted or deleted accidentally, half a line cleared by mistake, or some other major editing blunder is made. As the name implies, the "oops" command corrects most of these major editing errors. The "oops" command can be used to recover from the following commands:

- | | | |
|----|--------------------------|-------------|
| 1) | insert line command | (shift INS) |
| 2) | delete line command | (shift DEL) |
| 3) | hack command | (ctrl H) |
| 4) | to buffer command | (ctrl T) |
| 5) | from buffer command | (ctrl F) |
| 6) | copy from buffer command | (ctrl U) |
| 7) | split line command | (ctrl J) |

Screen Management

In addition to the commands available while in the edit mode, there are several other commands which are for use outside of the edit mode. Typically, these commands deal with entire screens at a time.

FLUSH (---)

When any changes are made to the current text window, the current screen is marked as having been changed. When leaving the edit mode using the "save" command, the current screen is sent to a set of internal FORTH buffers. These buffers are not written to disk until needed for other data. Thus, if no other screen is ever accessed, the buffers will never be saved to disk. The FLUSH command forces these buffers to be saved if they have been marked as being modified.

Example: FLUSH <ret>

EMPTY-BUFFERS (---)

Occasionally, screens are modified temporarily or by accident, and get marked as being modified. The EMPTY-BUFFERS command unmarks the internal FORTH buffers and fills them with zeroes so that "bad" data are not saved to disk. Zero filling the buffers ensures that the next access to any of the screens that were in the buffers will load the unadulterated copy from disk. The abbreviation MTB is included in the valFORTH system to make the use of this command easier.

Examples: EMPTY-BUFFERS <ret>
MTB <ret>

COPY (from to ---)

To duplicate a screen, the COPY command is used. The screen "from" is copied to the screen "to" but not flushed.

Example: 51 60 COPY <ret>

(Copies screen 51 to screen 60.)

CLEAR (scr# ---)

The CLEAR command fills the specified screen with blanks so that a clean edit can be started. The screen is then made current so that the L command can be used to enter the edit mode.

Example: 50 CLEAR <ret>

(Clears screen 50 and makes it current.)

CLEARs

(scr# #screens ---)

The CLEARs command is used to clear blocks of screens at a time. After user verification, it starts with the specified screen and clears the specified number of consecutive screens. The first screen cleared is made current so that the L command can be used to enter the edit mode.

Example: 25 3 CLEARs <ret>
 Clear from SCR 25
 to SCR 27 <Y/N> Y

(Screens 25-27 are cleared. Screen 25 is made current.)

SMOVE

(from to #screens ---)

The SMOVE command is a multiple screen copy command used for copying large numbers of consecutive screens at a time. User verification is required by this command to avoid disastrous loss of data. All screens to be copied are read into available memory and the user is prompted to initiate the copy. This allows the swapping of disks between moves to make disk transfers possible. The number of screens the SMOVE command can copy at a time is limited only by available memory.

Example: 50 60 5 SMOVE <ret>
 SMOVE from 50 thru 54
 to 60 thru 64 <Y/N> Y
 Insert source <RETURN> <ret>
 Insert dest. <RETURN> <ret>

(Transfers the specified screens.)

Editor Command Summary

Below is a quick reference list of all the commands which the video editor recognizes.

Entering the Edit Mode: (executed outside of the edit mode)

V (scr# ---)
Enter the edit mode and view the specified screen.

L (---)
Re-view the current screen.

WHERE (---)
Enter the edit mode and position the cursor over the word that caused a compilation error.

LOCATE cccc (---)
Enter the edit mode and position the cursor over the word defining "cccc".

LOCATOR (ON/OFF ---)
When ON, allows all words compiled until the next OFF to be locatable using the LOCATE command above.

#BUFS (#lines ---)
Sets the length (in lines) of the storage buffer. The default is five.

Cursor Movement:

(issued within the edit mode)

ctrl ^	Move cursor up one line, wrapping to the bottom line if moved off the top.
ctrl v	Move cursor down one line, wrapping to the top line if moved off the bottom.
ctrl <	Move cursor left one character, wrapping to the right edge if moved off the left.
ctrl >	Move cursor right one character, wrapping to the left edge if moved off the right.
RETURN	Position the cursor at the beginning of the next line.
TAB	Advance to next tabular column.

Editing Commands:

(issued within the edit mode)

ctrl INS	Insert one blank at cursor location, losing the last character on the line.
ctrl DEL	Delete character under cursor, closing the line.
shift INS	Insert blank line above current line, losing the last line on the screen.
shift DEL	Delete current cursor line, closing the screen.
ctrl I	Toggle insert-mode/replace-mode. (see full description of ctrl-I).
BACKS	Delete last character typed, if on the same line as the cursor.
ctrl H	Erase to end of line (Hack).

Buffer Management:

(issued within the edit mode)

- | | |
|--------|--|
| ctrl T | Delete current cursor line sending it to the edit buffer for later use. |
| ctrl F | Take the current buffer line and insert it above the current cursor line. |
| ctrl K | Copy current cursor line sending it to the edit buffer for later use. |
| ctrl U | Take the current buffer line and copy it to the current cursor line. |
| ctrl R | Roll the buffer making the next buffer line current. |
| ctrl B | Roll the buffer backwards making the previous buffer line on the screen current. |
| ctrl C | Clear the current buffer line and perform a ctrl-B. |

Note: The current buffer line is last line visible on the video display.

Changing Screens:

(issued within the edit mode)

- | | |
|--------|---|
| ctrl P | Display the previous screen saving all changes made to the current text window. |
| ctrl N | Display the next screen saving all changes made to the current text window. |
| ctrl S | Save the changes made to the current text window and end the edit session. |
| ctrl Q | Quit the edit session forgetting all changes made to current text window. |

Special Keys:

(issued within the edit mode)

- | | |
|--------|---|
| ESC | Do not interpret the next key typed as any of the commands above. Send it directly to the screen instead. |
| ctrl A | Put "-->", "==">", or erase the lower right-hand corner of the text window. |
| ctrl J | Split the current line into two lines at the point where the cursor is. |
| ctrl O | Corrects any major editing blunders. |

Screen Management:

(executed outside of the edit mode)

FLUSH

Save any updated FORTH screens to disk.

(---)

EMPTY-BUFFERS

Forget any changes made to any screens not yet FLUSHed to disk. Used in "losing" major editing mistakes. The abbreviation MTB is more commonly used.

(---)

COPY

Copies screen #from to screen #to.

(from to ---)

CLEAR

Blank fills specified screen. This performs the same functions as "WIPE" in Leo Brodie's book.

(scr# ---)

CLEARS

Blank fills the specified number of screens starting with screen scr#.

(scr# #screens ---)

SMOVE

Duplicate the specified number of screens Starting with screen number "from". Allows swapping of disks before saving screens to screen number "to".

(from to #screens ---)

STRING UTILITIES

The following collection of words describes the string utilities of the valFORTH Utilities Package. Strings have been implemented in the FORTH language in many different ways. Most implementations set aside space for a third stack -- a string stack. As strings are entered, they are moved (using CMOVE) to this stack. When strings are manipulated on this stack, many long memory moves are usually required. This method is typically much slower than the method implemented in valFORTH.

Rather than waste memory space with a third stack, valFORTH uses the already existing parameter stack. Unlike the implementation described above, valFORTH does not store strings on the stack. Rather, it stores the addresses of where the strings can be found.* Using this method, words such as SWAP , DUP , PICK , and ROLL can be used to manipulate strings. Routines such as string sorts which work on many strings at a time are typically much faster since addresses are manipulated rather than long strings. In practice, we have found few if any problems using this method of string representation.

String Glossary

For the purposes of this section, a string is defined to be a sequence of up to 255 characters preceded by a byte indicating its length. The first character of the string is referenced as character one. If the length of the string is zero, it has no characters and is called the "null" string. In stack notation, strings are represented by the symbol \$ and the address of the string is stored on the stack rather than the string itself*.

-TEXT addr1 n addr2 -- flag

The word -TEXT compares n characters at address1 with n characters at address2. Returns a false flag if the sequences match, true if they don't. Flag is positive if the character sequence at address1 is alphabetically greater than the one at address2. Flag is zero if the character sequences match, and is negative if the character sequence at address1 is alphabetically less than the one at address2.

-NUMBER addr -- d

-NUMBER functions identically to the standard FORTH word NUMBER with the only difference being that -NUMBER does not abort program execution upon an illegal conversion. -NUMBER takes the character string at addr and attempts to convert it to a double number. On successful conversion, the value d is returned with the status variable NFLG set to one. On unsuccessful conversion, a double number zero is returned with the variable NFLG set to zero. -NUMBER is pronounced "not number".

*Representing strings on the stack by their addresses is a very useful concept borrowed from MMS Forth (TRS-80), authored by Tom Dowling, and available from Miller Microcomputer Services, 617-653-6136.

NFLG -- addr

A variable used by -NUMBER that indicates whether the last conversion attempted was successful. NFLG is true if the conversion was successful; otherwise, it is false.

UMOVE addr1 addr2 n --

UMOVE is a "universal" memory move. It takes the block of memory n bytes long at addr1 and copies it to memory location addr2. UMOVE correctly uses either CMOVE or <CMOVE so that when a block of memory is moved onto part of itself, no data are destroyed.

" cccc" -- (at compile time)
cccc: -- addr (at run time)

If compiling, the sequence cccc (delimited by the trailing ") is compiled into the dictionary as a string:

l len | c | c | c |...| c |

All valFORTH strings are represented in this fashion. Since a single byte is used to store the length, a maximum string length of 255 is allowed. A string with 0 length is called a "null" string. At execution time, " puts the address in memory where the string is located onto the stack.

Note that " is IMMEDIATE. When executed outside of a colon definition, the string is not compiled into the dictionary, but is stored at PAD instead.

Example: " This is a string"

\$CONSTANT cccc \$ -- (at compile time)
cccc: -- \$ (at execution time)

Takes the string on top of the stack and compiles it into the dictionary with the name cccc. When cccc is later executed, the address of the string is pushed onto the stack.

Example: " Ready? <Y/N> " \$CONSTANT VERIFY

\$VARIABLE cccc n --
cccc: -- \$

Reserves space for a string of length n. When cccc is later executed, the address of the string is pushed onto the stack.

Example: 80 \$VARIABLE TEXTLINE

\$. \$ --

Takes the string on top of the stack and sends it to the current output device.

Example: " Hi there" \$. <ret> Hi there

\$! \$ addr --

Takes the string at second on stack and stores it at the address on top of stack.

Example: " Store me!" TEXTLINE \$!

\$+ \$1 \$2 -- \$3
 Takes \$2 and concatenates it with \$1, leaving \$3 at PAD.
 Example: " Santa " \$CONSTANT 1ST
 " Claus" \$CONSTANT LAST
 1ST LAST \$+
 \$. <ret> Santa Claus

LEFT\$ \$1 n -- \$2
 Returns the leftmost "n" characters of \$1 as \$2. \$2 is stored at PAD.
 Example: " They" 3 LEFT\$ \$. <ret> The

RIGHT\$ \$1 n -- \$2
 Returns the rightmost "n" characters of \$1 as \$2. \$2 is stored at PAD.
 Example: " mother" 5 RIGHT\$ \$. <ret> other

MID\$ \$1 n u -- \$2
 Returns \$2 of length u starting with the nth character of \$1. Recall that the first character of a string is numbered as one.
 Example: " Timeout" 3 2 MID\$ \$. <ret> me

LEN \$ -- len
 Returns the length of the specified string.

ASC \$ -- c
 Returns the ASCII value of the first character of the specified string.

\$COMPARE \$1 \$2 -- flag
 Compares \$1 with \$2 and returns a status flag. The flag is
 a) positive if \$1 is greater than \$2 or is equal to \$2, but longer,
 b) zero if the strings match and are the same length, and c) negative
 if \$1 less than \$2 or if they are equal and \$1 is shorter than \$2.

\$= \$1 \$2 -- flag
 Compares two strings on top of the stack and returns a status flag. The flag is true if the strings match and are equal in length, otherwise it is false.

\$< \$1 \$2 -- flag
 Compares two strings on top of the stack and returns a status flag. The flag is true if \$1 is less than \$2 or if \$1 matches \$2 but is shorter in length.

\$> \$1 \$2 -- flag
 Compares two strings on top of the stack and returns a status flag. The flag is true if \$1 is greater than \$2 or if \$1 matches \$2 but is longer in length.

SAVE\$ \$1 -- \$2
 As most string operations leave resultant strings at PAD, the word SAVE\$ is used to temporarily move strings to PAD+512 so that they can be manipulated without being altered in the process.
 Example: " Wash" SAVE\$ " ington" \$+

INSTR \$1 \$2 -- n
 Searches \$1 for first occurrence of \$2. Returns the character position in \$1 if a match is found; otherwise, zero is returned.
 Example: " FDCBA" \$CONSTANT GRADES
 GRADES " A" INSTR 1- . <ret> 4

CHR\$ c -- \$
 Takes the character "c" and makes it into a string of length one and stores it at PAD.

DVAL \$ -- d
 Takes numerical string \$ and converts it to a double length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.
 Example: " 123" DVAL D. <ret> 123

VAL \$ -- n
 Takes the numerical string \$ and converts it to a single length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.

DSTR\$ d -- \$
 Takes the double number d and converts it to its ASCII representation as \$ at PAD.
 Example: 123 DSTR\$ \$. <ret> 123

STR\$ n -- \$
 Takes the single length number n and converts it to its ASCII representation as \$ at PAD.

STRING\$ n \$1 -- \$2
 Creates \$2 as n copies of the first character of \$1.

#IN\$ n -- \$
 #IN\$ has three similar but different functions. If n is positive, it accepts a string of n or fewer characters from the terminal. If n is zero, it accepts up to 255 characters from the terminal. If n is negative, it returns only after accepting -n characters from the terminal. The resultant string is stored at PAD.

IN\$ -- \$
 Accepts a string of up to 255 characters from the terminal.

\$-TB \$1 -- \$2
 Removes trailing blanks from \$1 leaving new \$2.

\$XCHG \$1 -- \$2
 Exchanges the contents of \$1 with \$2.

ARRAYS and their COUSINS

All of the words described below create structures that are accessed in the same way, i.e., by putting the index or indices on the stack and then typing the structure's name. The differences are in the ways the structures are created.

The concept of the array should be known from BASIC. While in fig-FORTH there is no standard way to implement arrays and similar structures, there does exist a general consensus about how this should be done.

The point on which there is the most divergence of opinion is whether the first element in an array should be referred to by the index 0 or 1. We select 0 for the first index since this gives much cleaner code and makes more sense than 1 after you get used to it. (We've worked with it both ways.)

ARRAY and CARRAY, and 2ARRAY and 2CARRAY

The size of an array, specified when it is defined, is the number of elements in the array. In other words, an array defined by

```
8 ARRAY BINGO
```

will have 8 elements numbered 0 - 7.

To access an element of an array, do

```
n array-name
```

to get the address of the nth element on the stack. (You will not be told if the number n is not a legitimate index number for the array.) For example,

```
5 BINGO
```

will leave the address of element number 5 in BINGO on the stack. You can store to or fetch from this address as you require.

The word CARRAY defines a byte or character array. A c-array works the same as an array, except that you must use C@ and C! to manipulate single elements, rather than @ and !.

The words 2ARRAY and 2CARRAY each take two numbers during definition of a 2ARRAY or 2CARRAY, and 2ARRAYS and 2CARRAYS take two numbers to access an element. Note that when using a 2CARRAY named, say, CHESSBOARD, and a constant named ROOK, the two phrases

```
ROOK 4 6 CHESSBOARD C!
```

and

```
ROOK 6 4 CHESSBOARD C!
```

don't do the same thing. Also note that the phrase

8 8 2CARRAY CHESSBOARD

defines a 2CARRAY of $8 \times 8 = 64$ elements, with both indices running from 0 to 7.

When an ARRAY or a CARRAY is defined, the initial values of the elements are undefined.

TABLE AND CTABLE

A cousin of ARRAY is TABLE. Example: The phrase

```
TABLE THISLIST 14 , 18 , -34 , 16 ,
```

defines a table THISLIST of 4 elements. (The commas above are part of the code and must be included.) The number of elements does not have to be specified. The elements in THISLIST are accessed using the indices 0-3, the same as if it had been defined as an array. The word CTABLE works similarly, though using C, instead of , to compile in the numbers. Note that negatives won't be compiled in by a C, since in two's complement representation negative numbers always occupy the maximum number of bytes.

VECTOR and CVECTOR

The last array-type words in this package are CVECTOR and VECTOR. Vector is just another name for a list. These words are used when the elements of the array you want to create are on the stack, with the last element on top of the stack. You just put the number of elements on the stack and the VECTOR or CVECTOR, and the name you want to use. Example:

```
-3 8 127 899 -43      5 VECTOR    POSITIONS
```

creates an array named POSITIONS with 5 elements 0-4 with -3 in element 0 and -43 in element 4. CVECTOR works in a similar way.

EXAMPLES:

```
2 3 BINGO !
```

Stores the value 2 into element 3 of array BINGO.

```
2 THISLIST @
```

Will leave the value in element 2 of table THISLIST.

According to the definition of THISLIST above, this value will be -34.

```
3 POSITION @ . <cr> 899
```

ARRAY WORD GLOSSARY

ARRAY cccc, n -- (compiling)
cccc: m -- addr (executing)

When compiling, creates an array named cccc with n 16-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the array.

CARRAY cccc, n -- (compiling)
cccc: m -- addr (executing)

When compiling, creates a c-array named cccc with n 8-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the c-array.

TABLE cccc, -- (compiling)
cccc: m -- addr (executing)

When compiling, creates a table named cccc but does not allot space. Elements are compiled in directly with , (comma). When executing, takes one argument, m off the stack and, assuming 16-bit elements, leaves the address of element m of the table.

CTABLE cccc, -- (compiling)
cccc: m -- addr (executing)

When compiling, creates a c-table named cccc but does not allot space. Elements are compiled in directly with C, (c-comma). When executing, takes one argument, m off the stack and, assuming 8-bit elements, leaves the address of element m of the c-table.

X! n0 ... nN count addr --

Stores count 16-bit words, n0 thru nN into memory starting at addr, with n0 going into addr. Pronounced "extended store."

XC! b0 ... bN count addr --

Stores count 8-bit words, b0 thru bN into memory starting at addr, with b0 going into addr. Pronounced "extended c-store."

VECTOR cccc, n0 ... nN count -- (compiling)
cccc: m -- addr (executing)

When compiling, creates a vector named cccc with count 16-bit elements numbered 0-N. n0 is the initial value of element 0, nN is the initial value of element N, and so on. When executing, takes one argument, m, off the stack and leaves the address of element m on the stack.

CVECTOR cccc, b0 ... bN count -- (compiling)
cccc: m -- addr (executing)

When compiling, creates a c-vector named cccc with count 8-bit elements numbered 0-N. b0 is the initial value of element 0, bN is the initial value of element N, and so on. When executing, takes an argument, m, off the stack and leaves the address of element m on the stack.

CASE STRUCTURES

It often becomes necessary to make many tests upon a single number. Typically, this is accomplished by using a series of nested "DUP test IF" statements followed by a series of ENDIFs to terminate the IFs. This is arduous and very wasteful of memory. valFORTH contains four very powerful Pascal-type CASE statements which ease programming and conserve memory.

The CASE: structure

Format:

```
CASE: wordname
      word0
      word1
      ...
      wordN ;
```

The word CASE: creates words that expect a number from 0 to N on the stack. If the number is zero, word0 is executed; if the number is one, the word1 is executed; and so on. No error checks are made to ensure that the case number is a legal value.

Example:

```
: ZERO ." Zero" ;
: ONE  ." One"  ;
: TWO  ." Two"  ;
```

```
CASE: NUM
      ZERO
      ONE
      TWO ;
```

```
0 NUM <ret> Zero
1 NUM <ret> One
2 NUM <ret> Two
```

Note that any other number (e.g. 3 NUM) will crash the system.

The CASE Structure

Format:

```
      : wordname
      ...
      CASE
        word0
        word1
        ...
        wordN
      ( NOCASE wordnone )      (optional)
      CASEEND
      ... ;
```

The CASE...CASEEND structure is always used within a colon definition. Like CASE: above, it requires a number from 0 and N. However, unlike CASE: above, boundary checks are made so that an illegal case will do nothing. If the optional NOCASE clause is included then wordnone is executed if an "out of bounds" number is used.

Examples:

```
I)      : ZERO      ." Zero" ;
          : ONE       ." One"  ;
          : TWO       ." Two"  ;

          : CHECKNUM      ( n -- )
          CASE
            ZERO
            ONE
            TWO
          CASEEND ;

0 CHECKNUM <ret> Zero
1 CHECKNUM <ret> One
999 CHECKNUM <ret> (nothing happens)
2 CHECKNUM <ret> Two
```

```

II) : GRADEA      ." A" ;
    : GRADEB      ." B" ;
    : GRADEC      ." C" ;
    : GRADED      ." D" ;
    : OTHER       ." Failed" ;

```

```

DECIMAL
: GETGRADE      ( -- )
  KEY 65 -      (Convert A to 0, B to 1, etc)
  CASE
    GRADEA
    GRADEB
    GRADEC
    GRADED
  NOCASE OTHER
  CASEEND ;

```

```

GETGRADE <return and press A> A
GETGRADE <return and press B> B
GETGRADE <return and press F> Failed
GETGRADE <return and press D> D

```

The SEL Structure

Format:

```

: wordname
...
SEL                               (Select)
  n1 -> word0
  n2 -> word1
  ...
  nN > wordN
( NOSEL wordnone )              (optional)
SELEND
... ;

```


The SEL...SELEND structure is used when the "selection" numbers (n1 etc.) are not sequential. This structure is somewhat slower than either CASE or CASE: , but is much more general. SEL is typically used in operations such as table driver menus where single keystroke commands are used. The valFORTH video editor uses the SEL structure to implement the many editing keystroke commands.

Example:

```
I) : NICKEL ." nickel." ;
    : DIME ." dime." ;
    : QUARTER ." quarter." ;
    : 4BITS ." fifty cent piece." ;
    : SUSANB ." dollar" ;
    : BAD$$$ ." wooden nickel." ;

    : MONEY-NAME ( n -- )
      ." That is called a "
      SEL
        5 -> NICKEL
        10 -> DIME
        25 -> QUARTER
        50 -> 4BITS
        100 -> SUSANB
      NOSEL BAD$$$ ( this line is optional )
      SELEND ;

    5 MONEY-NAME <ret> That is called a nickel.
    33 MONEY-NAME <ret> That is called a wooden nickel.
    25 MONEY-NAME <ret> That is called a quarter.
```

The COND Structure

Format:

```
: wordname
...
COND
  condition0 << words0 >>
  condition1 << words1 >>
  ...
  conditionN << wordsn >>
( NOCOND wordsnone ) (optional)
CONDEND
... ;
```

Unlike the three previous CASE structures which test for equality, the COND structure bases its selection upon any true conditional test (e.g. if $n > 0$ then...) COND can also be used for range cases. The NOCOND clause is optional and is only executed if no other condition passes. Only the code of the first condition that passes will be executed.

Example:

```
: EXAM                                ( score -- grade )
COND
  90 >= << ." Grade of A" 4 >>
  80 >= << ." Grade of B" 3 >>
  70 >= << ." Grade of C" 2 >>
  60 >= << ." Grade of D" 1 >>
NOCOND ." Not too good" 0
CONDEND ;
```

Note that neither << nor >> are needed (nor allowed) around the "NOCOND" case. Also note that more than one word can be executed between the << and >> .

(intentionally left blank)

DOUBLE NUMBER EXTENSIONS

The following words extend the set of double number words to be as nearly identical as possible to the set in the book Starting FORTH. The exceptions are DVARIABLE and DCONSTANT which conform to the FIG standard by expecting initial values on the stack.

All of the single number operations comparable to the double number operations below were machine coded; all of the words below (with the exception of DVARIABLE) have high-level run time code and so are considerably slower than their single number counterparts.

DOUBLE NUMBER EXTENSION GLOSSARY

DVARIABLE cccc d --

cccc: -- addr

At compile time, creates a double number variable cccc with the initial value d.
At run time, cccc leaves the address of its value on the stack.

DCONSTANT cccc d --

cccc: -- d

At compile time, creates a double number constant cccc with the initial value d.
At run time, cccc leaves the value d on the stack.

0. -- 0.

A double number constant equal to double number zero.

1. -- 1.

A double number constant equal to double number one.

D- d1 d2 -- d3

Leaves d1-d2=d3.

D0= d -- flag

If d is equal to 0. leaves true flag; otherwise, leaves false flag.

D= d1 d2 -- flag

If d1 equals d2, leaves true flag; otherwise, leaves false flag.

D0< d -- flag

If d is negative, leaves true flag; otherwise, leaves false flag.

D< d1 d2 -- flag

If d1 is less than d2, leaves true flag; otherwise, leaves false flag.

D> d1 d2 -- flag

If d1 is greater than d2, leaves true flag; otherwise, leaves false flag.

DMIN d1 d2 -- d3
Leaves the minimum of d1 and d2.

DMAX d1 d2 -- d3
Leaves the maximum of d1 and d2.

D>R d --
Sends the double number at top of stack to the return stack.

DR> -- d
Pulls the double number at top of the return stack to the stack.

D, d --
Compiles the double number at top of stack into the dictionary.

DU< ud1 ud2 -- flag
If the unsigned double number ud1 is less than the unsigned double number ud2, leaves a true flag; otherwise, leaves a false flag.

M+ d1 n -- d2
Converts n to a double number and then sums with d1.

HIGH RESOLUTION TEXT OUTPUT

Occasionally, the need arises to print text in high resolution graphic displays (8 GR.). The following set of words explains how Graphic Characters can be used in valFORTH programs. The Graphic-Character output routines are designed to function identically to the standard FORTH output operations. There is an invisible cursor on the high resolution page which always points to where the next graphic-character will be printed. As with normal text output, this cursor can be repositioned at any time and in various ways. Because of the nature of hi-res printing, this cursor can also be moved vertically by partial characters. This allows for super/subscripting, overstriking, and underlining. Multiple character fonts on the same line are also possible.

- GCINIT --
 Initializes the graphic character output routines. This must be executed prior to using any other hi-res output words.
- GC. n --
 Displays the single length number n at the current hi-res cursor location.
- GC.R n1 n2 --
 Displays the single length number n1 right-justified in a field n2 graphic characters wide. See .R .
- GCD.R d n --
 Displays the double length number d right-justified in a field n graphic characters wide. See D.R .
- GCEMIT c --
 Displays the text character c at the current hi-res cursor location. Three special characters are interpreted by GCEMIT . The up arrow (↑) forces text output into the superscript mode; the down arrow (↓) forces the text into the subscript mode; and the left arrow (←) performs a GCBKS command (described below). See OSTRIKE below; also see EMIT.
- GCLLEN addr n -- len
 Scans the first n characters at addr and returns the number of characters that will actually be displayed on screen. This is typically used to find the true length of a string that contains any of the non-printing special characters described in GCEMIT above. Used principally to aid in centering text, etc.
- GCR --
 Repositions the hi-res cursor to the beginning of the next hi-res text line. See CR .
- GCLS --
 Clears the hi-res display and repositions the cursor in the upper lefthand corner.

GCSPACE --
Sends a space to the graphic character output routine. See SPACE .

GCSPACES n --
Sends n spaces to the graphic character output routine. See SPACES .

GCTYPE addr n --
Sends the first n characters at addr to the graphic character output routine. See TYPE .

GC" cccc" --
Sends the character string cccc (delimited by ") to the graphic character output routine. If in the execution mode, this action is taken immediately. If in the compile mode, the character string is compiled into the dictionary and printed out only when executed in the word that uses it. See ." .

GCBKS --
Moves the hi-res cursor back one character position for overstriking or underlining.

GCPOS horz vert --
Positions the hi-res cursor to the coordinates specified. Note that the upper lefthand corner is 0,0.

GC\$. addr --
Sends the string found at addr and preceded by a count byte to the graphic character output routine. See \$. .

SUPER --
Forces the graphic character output routine into the superscript mode (or out of the subscript mode). See VMI below. May be performed within a string by the ^ character.

SUB --
Forces the graphic character output routine into the subscript mode (or out of the superscript mode). See VMI below. May be performed within a string by the v character.

VMI n --
Each character is eight bytes tall. The VMI command sets the number of eighths of characters to scroll up or down when either a SUPER or SUB command is issued. Normally, 4 VMI is used to scroll 4/8 or half a character in either direction.

VMI# -- addr
A variable set by VMI.

OSTRIKE ON or OFF --
The GCEDIT command has two separate functions. If OSTRIKE (overstrike) option is OFF, the character output will replace the character at the current cursor position. This is the normal method of output. If the OSTRIKE option is ON, the new character is printed over top of the previous character giving the impression of an overstrike. This allows the user to underline text and create new characters: Example: To do underline, a value of, say, 2 should be used with VMI, and then the v character added in the string before the underline character.

GCBAS

-- addr

A variable which contains the address of the character set displayed by GCEMIT. To change character sets, simply store the address of your new character set into this variable.

GCLFT

-- addr

A variable which holds the column position of the left margin. Normally two, this can be changed to obtain a different display window.

GCRGT

-- addr

A variable which holds the column position of the right margin. Normally 39, this can be changed to obtain a different display window.

(intentionally left blank)

MISCELLANEOUS UTILITIES

This is a grab-bag of useful words. Here they are...

XR/W #secs addr blk flag --

"Extended read-write." The same as R/W except that XR/W accepts a sector count for multiple sector reads and writes. Starting at address addr and block blk, read (flag true) or write (flag false) #secs sectors from or to disk.

SMOVE org des count --

Move count screens from screen # org to screen # dest.

The primary disk rearranging word, also used for moving sequences of screens between disks. This is a smart routine that uses all memory available below the current GR.-generated display list, with prompts for verification and disk swap if desired. See valFORTH Editor 1.1 documentation for further details.

LOADS start count --

Loads count screens starting from screen # start. This word is used if you want to use words that are not chained together by --> 's. It will stop loading if a CONSOLE button is held down when the routine finishes loading its present screen.

THRU start finish -- start count

Converts two range numbers to a start-count format. Example:

120 130 THRU PLISTS

will print screens 120 thru 130.

SEC n --

Provides an n second delay. Uses a tuned do-loop.

MSEC n --

Provides an n millisecond delay. (approx)

Uses a tuned do-loop.

H->L n -- b

Moves the high byte of n to the low byte and zero's the high byte, creating b. Machine code.

L->H n1 -- n2

Moves the low byte of n1 to the high byte and zero's the low byte, creating n2. Machine code.

H/L n1 -- n1(hi) n1(lo)

Split top of stack into two stack items: New top of stack is low byte of old top of stack. New second on stack is old top of stack with low byte zeroed.

Example: HEX 1234 H/L .S <cr> 1200 0034

BIT b -- n

Creates a number n that has only its bth bit set. The bits are numbered 0-15, with zero the least significant. Machine code.

?BIT n b -- f

Leaves a true flag if the bth bit of n is set. Otherwise leaves a false flag.

TBIT n1 b -- n2

Toggles the bth bit of n1, making n2.

SBIT n1 b -- n2

Sets the bth bit of n1, making n2.

RBIT n1 b -- n2

Resets the bth bit of n1, making n2.

STICK n -- horiz vert

Reads the nth stick (0-3) and resolves the setting into horizontal and vertical parts, with values from -1 to +1. -1 -1 means up and to the left.

PADDLE n1 -- n2

Reads the nth paddle (0-7) and returns its value n2. Machine code.

ATTRACT f --

If the flag is true, the attract mode is initiated. If the flag is false, the attract mode is terminated.

NXTATR --

If the system is in the attract mode, this command cycles to the next color setup in the attract sequence. Disturbs the timer looked at by 16TIME.

HLDATR --

If the system is in attract mode, zero's fast byte of the system timer so that attract won't cycle to next color setup for at least four seconds or until system timer is changed, say by NXTATR. Disturbs the timer looked at by 16TIME.

16TIME -- n

Returns a 16 bit timer reading from the system clock at locations 19 and 20, decimal. This clock is updated 60 times per second, with the fast byte in 20. Machine code, not fooled by carry.

8RND -- b

Leaves one random byte from the internal hardware. Machine code.

16RND -- n

Leaves one random word from the internal hardware. Machine code with 20 cycle extra delay for rerandomization.

CHOOSE u1 -- u2

Randomly choose an unsigned number u2 which is less than u1.

CSHUFL addr n --
Randomly rearrange n bytes in memory, starting at address addr.
Pronounced "c-shuffle."

SHUFL addr n --
Randomly rearrange n words in memory, starting at address addr. Pronounced "shuffle." SHUFL may also be used to shuffle items directly on the stack by doing SP@ n SHUFL.

H, n --
See DEBUG Glossary.

A. addr --
Print the ASCII character at addr, or if not printable, print a period.
(Used by DUMP).

DUMP addr n --
Starting at addr, dump at least n bytes (even multiple of 8) as ASCII and hex. May be exited early by pressing a CONSOLE button.

BLKOP system use only

BXOR addr count b --
Starting at address addr, for count bytes, perform bit-wise exclusive or with byte b at each address. Useful for toggling an area of display memory to inverse video or a different color, and for other purposes. For instance, in 0 GR., do

DCX 88 @ 280 128 BXOR

Then do Shift-Clear to clear the screen. Pronounced "block ex or."

BAND addr count b --
Starting at address addr, for count bytes, perform bit-wise AND with byte b at each address. Applications similar to BXOR.
Pronounced "block and."

BOR addr count b -
Starting at address addr, for count bytes, perform bit-wise or with byte b at each address. Applications similar to BXOR.
Pronounced "block or."

STRIG n -- flag
Reads the button of joystick n (0-3). Leaves a true flag if the button is pressed, a false flag if it isn't.

PTRIG n -- flag
Reads the button of paddle n (0-7). Leaves a true flag if the button is pressed, a false flag if it isn't.

(intentionally left blank)

TRANSIENTS

One of the more annoying parts about common releases of FORTH concerns the FORTH machine code assemblers. On the positive side, FORTH-based assemblers can be extraordinarily smart and easy to use interactively, and can compile on the fly as you type, rather than in multiple-pass fashion. (The 6502 assembler provided with valFORTH is a good example of a smart, structured, FORTH-based assembler.) On the other hand, since the assembler loads into the dictionary one usually sacrifices between 3 and 4K of memory on a utility that is only a compilation aid, and is not used during execution. With the utility described below, however, you can use the assembler and then remove it from the dictionary when you're finished with it.

In the directory of the Utilities/Editor disk (screen 170) you will find a heading of Transients. Loading this screen brings in three words: TRANSIENT, PERMANENT, and DISPOSE, and a few variables. It also defines a new area of memory called the Transient area. This area is used to load utilities like the assembler, certain parts of case statements, and similar constructs, that have one characteristic in common: They have compile-time behavior only, and are not used at run-time. An example will help make clear the sequence of operations. You may recall that on the valFORTH disk, in order to load floating point words you needed the assembler. Let's make a disk that has floating point but no assembler:

- * Boot your valFORTH disk. It can be the bare system, or your normal programming disk if it doesn't have the assembler already in it.
- * Insert your Utilities/Editor disk, find the Transient section in the directory, and load it.
- * Do MTB (EMPTY-BUFFERS) and swap in your valFORTH disk. (It is a VERY good idea to get into the habit of doing MTB before swapping disks.) Find the assembler in the directory, but before you load it, do TRANSIENT to cause it to be loaded into the transient dictionary area, in high memory. Now go ahead and load the assembler. When it is loaded, do PERMANENT so that the next entries will go into the permanent dictionary area, which is back where you started.
- * Now find and load the floating point words.
- * Finally, do DISPOSE to pinch off the links that tie the transient area (with the assembler in it) to the permanent dictionary, with the floating point words in it. Do a VLIST or two to prove it to yourself. (Note that there are about a half-dozen words in the assembler vocabulary in the kernel. These were in the dictionary on boot up and are not affected by DISPOSE.)

You can derive great benefit from the simple recipe above, and if you study the Transient code a bit, you may learn even more. We offer several comments:

* In the case of the above recipe, you didn't actually have to do PERMANENT and TRANSIENT because the assembler source code checks at the front to see if TRANSIENT exists, and does it if so. At the end it checks to see if PERMANENT exists, and does it if so. This conditional execution is accomplished with the valFORTH construct

'()()

which is described in valFORTH documentation. Take a look at the assembler source code to see how this is done.

* If you want to do assembly on more than one section of code, you needn't DISPOSE until you really finished with the assembler; or, if you have DISPOSED of the assembler, you can bring it back in later without harm, by the same method. You can also code high-level definitions, and then more assembly code, and so on, and only do DISPOSE when you were finished. Be sure to do DISPOSE before SAVE or AUTO, however, because either your system will crash or your SAVE'd or AUTO'd program won't work.

The situation is slightly different with "case" words, since if you bring them in more than once you'll get duplicate names on the run-time words like (SEL), (CASE) and CASE:, which uses extra space and defeats the purpose of Transients.

* If you use the Transient structures for other purposes, remember only to send code that is not used at run-time to the transient area. As an example of this distinction, look at the code for the "case" words on the valFORTH disk. Note that the '()() construct is again used, but that some of the parts of the case constructs, for instance (SEL), stay in the permanent dictionary. That is because (SEL) actually ends up in the compiled code, while SEL does not.

* Look at the beginning of the code for the Transient structures, and notice that the Transient area has been set up 4000 bytes below the display list. (The byte just below the display list in normal modes is pointed to by memory location 741 decimal, courtesy of the Atari OS.) This is usually a good place if only the 0 Graphics mode is used. (8 GR., for example, will over-write this area, crashing the system.) After DISPOSE is executed, this area is freed for other purposes. If you want to use a different area for Transients, just substitute your address into the source code on the appropriate screen. Remember that you must leave enough room for whatever will go into the Transient dictionary, and that NOTHING else must write to the area until you have cleared it out with DISPOSE. (This includes SMOVE, DISKCOPY1, DISKCOPY2, etc.)

***** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE *****

In the above example, 4000 bytes have been set aside for the Transient area just below the 0 GR. display list. This amount of memory will generally hold the assembler and some case statement compiling words. REMEMBER that if you have relocated the buffers (see the section on Relocating Buffers) to this area as well, you will have a collision, and a crashed system in short order.

To cure this, simply locate the Transient area 2113 bytes lower in memory so that there will be no overlap.

***** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE *****

ACKNOWLEDGEMENT

Various implementations of the Transient concept have appeared. valFORTH adopts the names TRANSIENT, PERMANENT, and DISPOSE from a public domain article by Phillip Wasson which appeared in FORTH DIMENSIONS volume III no. 6. The Transient structure implemented in the article has been altered somewhat in the valFORTH implementation to allow DISPOSE to dispose of the entire Transient structure, including DISPOSE itself, thus rendering the final product perfectly clean.

FORTH DIMENSIONS is a publication available through FIG (address listed elsewhere) and can be a valuable source of information and ideas to the advanced FORTH programmer.

EDITOR/UTILITIES SUPPLIED SOURCE

Screen: 36

```

0 ( Transients:  setup          )
1 BASE @ DCX
2
3 HERE
4
5
6 741 @ 4000 - DP !
7 ( SUGGESTED PLACEMENT OF TAREA )
8
9
10 HERE CONSTANT TAREA
11 0 VARIABLE TP
12 1 VARIABLE TPFLAG
13 VARIABLE OLDDP
14
15 ==>

```

Screen: 39

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 37

```

0 ( Xsients: TRANSIENT PERMANENT )
1
2 : TRANSIENT          ( -- )
3 TPFLAG @ NOT
4 IF HERE OLDDP ! TP @ DP !
5 1 TPFLAG !
6 ENDIF ;
7
8 : PERMANENT          ( -- )
9 TPFLAG @
10 IF HERE TP ! OLDDP @ DP !
11 0 TPFLAG !
12 ENDIF ;
13
14
15 --->

```

Screen: 40

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 38

```

0 ( Transients:  DISPOSE          )
1 : DISPOSE  PERMANENT
2 CR ." Disposing..." VOC-LINK
3 BEGIN DUP 0 53279 C!
4 BEGIN @ DUP TAREA U<
5 UNTIL DUP ROT ! DUP 0=
6 UNTIL DROP VOC-LINK @
7 BEGIN DUP 4 -
8 BEGIN DUP 0 53279 C!
9 BEGIN PFA LFA @ DUP TAREA U<
10 UNTIL
11 DUP ROT PFA LFA ! DUP 0=
12 UNTIL DROP @ DUP 0=
13 UNTIL DROP [COMPILE] FORTH
14 DEFINITIONS ." Done" CR ;
15 PERMANENT BASE !

```

Screen: 41

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 42

```

0 ( Utils: CARRAY ARRAY )
1 BASE @ HEX
2 : CARRAY ( cccc, n -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ALLOT
5 ;CODE CA C, CA C, 18 C,
6 A5 C, W C, 69 C, 02 C, 95 C,
7 00 C, 98 C, 65 C, W 1+ C,
8 95 C, 01 C, 4C C,
9 ' + ( CFA @ ) , C;
10
11 : ARRAY ( cccc, n -- )
12 CREATE SMUDGE ( cccc: n -- a )
13 2* ALLOT
14 ;CODE 16 C, 00 C, 36 C, 01 C,
15 4C C, ' CARRAY 08 + , C; ==>

```

Screen: 43

```

0 ( Utils: CTABLE TABLE )
1
2 : CTABLE ( cccc, -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ;CODE
5 4C C, ' CARRAY 08 + , C;
6
7 : TABLE ( cccc, -- )
8 CREATE SMUDGE ( cccc: n -- a )
9 ;CODE
10 4C C, ' ARRAY 0A + , C;
11
12
13
14
15 -->

```

Screen: 44

```

0 ( Utils: 2CARRAY 2ARRAY )
1
2 : 2CARRAY ( cccc, n n -- )
3 <BUILDS ( cccc: n n -- a )
4 SWAP DUP , * ALLOT
5 DOES>
6 DUP >R @ * + R> + 2+ ;
7
8 : 2ARRAY ( cccc, n n -- )
9 <BUILDS ( cccc: n n -- a )
10 SWAP DUP , * 2* ALLOT
11 DOES>
12 DUP >R @ * + 2* R> + 2+ ;
13
14
15 ==>

```

Screen: 45

```

0 ( Utils: XC! X! )
1
2 : XC! ( n0...nm cnt addr -- )
3 OVER 1- + >R 0
4 DO J I - C!
5 LOOP R> DROP ;
6
7 : X! ( n0...nm cnt addr -- )
8 OVER 1- 2* + >R 0
9 DO J I 2* - !
10 LOOP R> DROP ;
11
12 ( Caution: Remember limitation
13 ( on stack size of 30 values
14 ( because of OS conflict. )
15 -->

```

Screen: 46

```

0 ( Utils: CVECTOR VECTOR )
1
2 : CVECTOR ( cccc, cnt -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 HERE OVER ALLOT XC!
5 ;CODE
6 4C C, ' CARRAY 08 + , C;
7
8 : VECTOR ( cccc, cnt -- )
9 CREATE SMUDGE ( cccc: n -- a )
10 HERE OVER 2* ALLOT X!
11 ;CODE
12 4C C, ' ARRAY 0A + , C;
13
14
15 BASE !

```

Screen: 47

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 48

```

0 ( Utils: HIDCHR NOKEY CURSOR)
1 BASE @ DCX
2
3 '( CASE )( 28 KLOAD )
4
5 : HIDCHR ( -- )
6 65535 94 ! ;
7
8 : NOKEY ( -- )
9 255 764 C! ; )
10
11 : CURSOR ( f -- )
12 0= 752 C!
13 28 EMIT 29 EMIT ;
14
15 ==>

```

Screen: 49

```

0 ( Utils: INKEY$ )
1 DCX
2 : (INKEY$) ( c -- )
3 702 C! NOKEY ;
4
5 : INKEY$ ( -- c )
6 764 C@
7 COND
8 252 = << 128 (INKEY$) 0 >>
9 191 > << 0 >>
10 188 = << 0 >>
11 124 = << 64 (INKEY$) 0 >>
12 60 = << 0 (INKEY$) 0 >>
13 39 = << 0 >>
14 NOCOND KEY
15 CONDEND ; -->

```

Screen: 50

```

0 ( Utils: -Y/N )
1
2 : -Y/N ( -- f )
3 BEGIN KEY
4 COND
5 89 = << 1 1 >>
6 78 = << 0 1 >>
7 NOCOND
8 0
9 CONDEND
10 UNTIL ;
11
12
13
14
15 ==>

```

Screen: 51

```

0 ( Utils: Y/N -RETURN RETURN )
1
2 : Y/N ( -- f )
3 ." <Y/N> " -Y/N DUP
4 IF 89 ELSE 78 ENDIF
5 EMIT SPACE ;
6
7 : -RETURN ( -- )
8 BEGIN KEY 155 = UNTIL ;
9
10 : RETURN ( -- )
11 ." <RETURN> " -RETURN ;
12
13
14
15 BASE !

```

Screen: 52

```

0 ( Screen code conversion words )
1
2 BASE @ HEX
3
4 CODE >BSCD ( a a n -- )
5 A9 C, 03 C, 20 C, SETUP ,
6 HERE C4 C, C2 C, D0 C, 07 C,
7 C6 C, C3 C, 10 C, 03 C, 4C C,
8 NEXT , B1 C, C6 C, 48 C,
9 29 C, 7F C, C9 C, 60 C, B0 C,
10 0D C, C9 C, 20 C, B0 C, 06 C,
11 18 C, 69 C, 40 C, 4C C, HERE
12 2 ALLOT 38 C, E9 C, 20 C, HERE
13 SWAP ! 91 C, C4 C, 68 C, 29 C,
14
15 ==>

```

Screen: 53

```

0 ( Screen code conversion words )
1
2 80 C, 11 C, C4 C, 91 C, C4 C,
3 C8 C, D0 C, D3 C, E6 C, C7 C,
4 E6 C, C5 C, 4C C, , C;
5
6 CODE BSCD> ( a a n -- )
7 A9 C, 03 C, 20 C, SETUP ,
8 HERE C4 C, C2 C, D0 C, 07 C,
9 C6 C, C3 C, 10 C, 03 C, 4C C,
10 NEXT , B1 C, C6 C, 48 C,
11 29 C, 7F C, C9 C, 60 C, B0 C,
12 0D C, C9 C, 40 C, B0 C, 06 C,
13 18 C, 69 C, 20 C, 4C C, HERE
14 2 ALLOT 38 C, E9 C, 40 C, HERE
15 -->

```

Screen: 54

```
0 ( Screen code conversion words )
1
2 SWAP ! 91 C, C4 C, 68 C, 29 C,
3 80 C, 11 C, C4 C, 91 C, C4 C,
4 C8 C, D0 C, D3 C, E6 C, C7 C,
5 E6 C, C5 C, 4C C, ,
6
7
8 : >SCD SP@ DUP 1 >BSCD ;
9 : SCD> SP@ DUP 1 BSCD> ;
10
11
12
13
14
15 BASE !
```

Screen: 55

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 56

```
0 ( Case Statements: CASE )
1 BASE @ DCX
2 '( PERMANENT PERMANENT )( )
3 : (CASE)
4 R C@ MIN -1 MAX 2*
5 R 3 + + @EX
6 R C@ 2* 5 + R) + >R ;
7 '( TRANSIENT TRANSIENT )( )
8 : CASE
9 ?COMP COMPILE (CASE)
10 HERE 0 C,
11 COMPILE NOOP 6 ; IMMEDIATE
12
13 : NOCASE
14 6 ?PAIRS 7 ; IMMEDIATE
15 ==>
```

Screen: 57

```
0 ( Case statements: CASE
1
2 : CASEEND
3 DUP 6 =
4 IF
5 DROP COMPILE NOOP
6 ELSE
7 7 ?PAIRS
8 ENDIF
9 HERE 2- @ OVER 1+ !
10 HERE OVER -
11 5 - 2/ SWAP C! ; IMMEDIATE
12
13 '( PERMANENT PERMANENT )( )
14
15 -->
```

Screen: 58

```
0 ( Case statements: SEL )
1
2 '( PERMANENT PERMANENT )( )
3 : (SEL)
4 R 1+ DUP 2+ DUP R C@
5 2* 2* + R) DROP DUP >R SWAP
6 DO I @ 3 PICK =
7 IF I 2+ SWAP DROP LEAVE
8 ENDIF
9 4 /LOOP SWAP DROP @EX ;
10
11 '( TRANSIENT TRANSIENT )( )
12 : SEL ?COMP
13 ?LOADING COMPILE (SEL) HERE
14 0 C, COMPILE NOOP [COMPILE] [
15 8 ; IMMEDIATE ==>
```

Screen: 59

```
0 ( Case statements: SEL )
1
2 : NOSEL
3 8 ?PAIRS [COMPILE] ' CFA
4 OVER 1+ ! 8 ; IMMEDIATE
5
6 : ->
7 SWAP 8 ?PAIRS , DUP C@ 1+
8 OVER C! [COMPILE] '
9 CFA , 8 ; IMMEDIATE
10
11 : SELEND
12 8 ?PAIRS
13 DROP [COMPILE] ] ; IMMEDIATE
14 '( PERMANENT PERMANENT )( )
15 -->
```

Screen: 60

```
0 ( Case statements: COND )
1 '( TRANSIENT TRANSIENT )( )
2 : COND
3 0 COMPILE DUP ; IMMEDIATE
4
5 : <<
6 1+ [COMPILE] IF
7 COMPILE DROP ; IMMEDIATE
8
9 : >>
10 [COMPILE] ELSE COMPILE
11 DUP ROT ; IMMEDIATE
12
13 : NOCOND
14 COMPILE 2DROP ; IMMEDIATE
15 '( PERMANENT PERMANENT )( ) ==>
```

Screen: 63

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 61

```
0 ( Case statements: COND )
1
2 '( TRANSIENT TRANSIENT )( )
3
4 : CONDEND
5 0 DO
6 [COMPILE] ENDIF
7 LOOP ; IMMEDIATE
8
9 '( PERMANENT PERMANENT )( )
10
11
12
13
14
15 -->
```

Screen: 64

```
0 ( ValFORTH Video editor V1.1 )
1
2 BASE @ DCX
3
4 '( XC! )( 21 KLOAD )
5 '( HIDCHR )( 24 KLOAD )
6 '( )BSCD )( 26 KLOAD )
7
8
9
10
11
12
13
14
15 ==>
```

Screen: 62

```
0 ( Case statements: CASE: )
1
2 : CASE:
3 <BUILDS
4 SMUDGE !CSP
5 [COMPILE] ]
6 DOES>
7 SWAP 2* + @EX ;
8
9
10
11
12
13
14
15 BASE !
```

Screen: 65

```
0 ( ValFORTH Video editor V1.1 )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->
```

Screen: 66

```

0 ( ValFORTH Video editor V1.1 )
1
2 VOCABULARY EDITOR IMMEDIATE
3 EDITOR DEFINITIONS
4
5 0 VARIABLE XLOC ( X coord. )
6 0 VARIABLE YLOC ( Y coord. )
7 0 VARIABLE INSRT ( insert on? )
8 0 VARIABLE LSTCHR ( last key )
9 0 VARIABLE ?BUFSM ( buf same? )
10 0 VARIABLE ?PADSM ( PAD same? )
11 0 VARIABLE ?ESC ( coded char? )
12 0 VARIABLE TBLK ( top block )
13
14
15 ==>

```

Screen: 69

```

0 ( ValFORTH Video editor V1.1 )
1
2 : UPCUR ( -- )
3 CBLANK YLOC @
4 1 - DUP 0<
5 IF DROP 15 ENDIF
6 YLOC ! CSHOW ;
7
8
9 : DNCUR ( -- )
10 CBLANK YLOC @
11 1 + DUP 15 )
12 IF DROP 0 ENDIF
13 YLOC ! CSHOW ;
14
15 -->

```

Screen: 67

```

0 ( ValFORTH Video editor V1.1 )
1
2 0 VARIABLE LNFLG ( oops flag )
3 4 ARRAY UPSTAT ( update map )
4 15 CONSTANT 15
5 32 CONSTANT 32
6 128 CONSTANT 128
7 5 32 * CONSTANT BLEN
8
9 : LMOVE 32 CMOVE ;
10 : BOL 88 @ YLOC @ 1+ 32 * + ;
11 : SBL 88 @ 544 + ;
12 : PBL PAD 544 + ;
13 : PBL PBL BLEN + 32 - ;
14 : !SCR 88 @ 32 + PAD 512 BSCD) ;
15 -->

```

Screen: 70

```

0 ( ValFORTH Video editor V1.1 )
1
2 : LFCUR ( -- )
3 CBLANK XLOC @
4 1 - DUP 0< ( AT L-SIDE?)
5 IF DROP 31 ENDIF ( FIX IF SO )
6 XLOC ! CSHOW ;
7
8 : RTCUR ( -- )
9 CBLANK XLOC @
10 1+ DUP 31 ) ( AT R-SIDE?)
11 IF DROP 0 ENDIF ( FIX IF SO )
12 XLOC ! CSHOW ;
13
14 : EDMRK
15 1 YLOC @ 4 / UPSTAT ! ; ==>

```

Screen: 68

```

0 ( ValFORTH Video editor V1.1 )
1
2 : CURLOC ( -- )
3 BOL XLOC @ + ; ( SCR ADDR )
4
5 : CSHOW ( -- )
6 CURLOC DUP ( GET SCR ADDR )
7 C@ 128 OR ( INVERSE CHAR )
8 SWAP C! ; ( STORE ON SCR )
9
10 : CBLANK ( -- )
11 CURLOC DUP ( GET SCR ADDR )
12 C@ 127 AND ( STRIP MSB )
13 SWAP C! ; ( STORE IT )
14
15 ==>

```

Screen: 71

```

0 ( ValFORTH Video editor V1.1 )
1
2 : INTGL ( -- )
3 INSRT @ 0= ( TOGGLE THE )
4 INSRT ! ; ( INSRT FLAG )
5
6 : NXTLN ( -- )
7 CBLANK 0 XLOC !
8 CSHOW DNCUR ;
9
10 : CLREOL ( -- )
11 CBLANK !SCR
12 1 LNFLG ! CURLOC ( CLEAR )
13 32 XLOC @ - ( TO END )
14 ERASE CSHOW ( OF LINE )
15 EDMRK ; -->

```

Screen: 72

```

0 ( ValFORTH Video editor V1.1 )
1
2 : HMCUR ( -- )
3 CBLANK 0 XLOC !
4 0 YLOC ! CSHOW ;
5
6 : BYTINS CBLANK ( -- )
7 XLOC @ 31 ( SPREAD LN )
8 IF
9 CURLOC DUP 1+ ( FROM, TO )
10 31 XLOC @ - ( # CHARS )
11 CMOVE ( MOVE IT )
12 ENDIF
13 0 CURLOC C! ( CLEAR OLD )
14 CSHOW EDMRK ; ( CHARACTER )
15 ==>

```

Screen: 75

```

0 ( ValFORTH Video editor V1.1 )
1
2 : LNDEL ( -- )
3 CBLANK 3 LNFLG ! !SCR
4 4 YLOC @ 4 /
5 DO 1 I UPSTAT ! LOOP
6 YLOC @ 15 (
7 IF BOL ( FROM )
8 DUP 32 + SWAP ( TO )
9 15 YLOC @ - 32 * ( # CH )
10 CMOVE
11 ENDIF
12 BOL 15 YLOC @ -
13 32 * + 32 ERASE
14 CSHOW EDMRK ;
15 -->

```

Screen: 73

```

0 ( ValFORTH Video editor V1.1 )
1
2 : BYTDEL ( -- )
3 CBLANK ( CLOSE LINE )
4 XLOC @ 31 (
5 IF
6 CURLOC DUP ( FROM ADDR )
7 1+ SWAP ( TO ADDR )
8 31 XLOC @ - ( # CHARS )
9 CMOVE ( MOVE IT )
10 ENDIF
11 0 CURLOC ( BLANK OUT )
12 31 XLOC @ - + C! ( CHAR AT )
13 CSHOW EDMRK ; ( END OF LN )
14
15 -->

```

Screen: 76

```

0 ( ValFORTH Video editor V1.1 )
1
2 : BFSHW ( -- )
3 PBL 128 - ( F, T )
4 SBL 160 CMOVE ; ( # MOVE )
5
6 : BFROT ( -- )
7 PBL DUP
8 BLEN + LMOVE
9 PBL DUP 32 +
10 SWAP BLEN 32 -
11 CMOVE PBL 32 +
12 PBL LMOVE
13 BFSHW ;
14
15 ==>

```

Screen: 74

```

0 ( ValFORTH Video editor V1.1 )
1
2 : LNINS ( -- )
3 CBLANK 2 LNFLG ! !SCR
4 4 YLOC @ 4 /
5 DO 1 I UPSTAT ! LOOP
6 YLOC @ 15 (
7 IF
8 BOL DUP 32 +
9 15 YLOC @ - 32 *
10 CMOVE
11 ENDIF
12 BOL 32 ERASE
13 CSHOW EDMRK ;
14
15 ==>

```

Screen: 77

```

0 ( ValFORTH Video editor V1.1 )
1
2 : <BFROT ( -- )
3 PBL DUP
4 32 + LMOVE
5 PBL DUP 32 +
6 BLEN 32 - CMOVE
7 PBL DUP BLEN +
8 SWAP LMOVE
9 BFSHW ;
10
11 : BFCLR ( -- )
12 PBL 32 ERASE
13 <BFROT ;
14
15 -->

```

Screen: 78

```

0 ( ValFORTH Video editor V1.1 )
1
2 : BFCPY ( -- )
3 CBLANK BFROT ( BRING LN )
4 BOL PBLL ( DOWN TO )
5 LMOVE BFSHW ( BUFFER & )
6 CSHOW ; ( ROTATE )
7
8 : >BFNXT BFCPY NXTLN ; ( -- )
9
10 : >BFLN BFCPY LNDEL ; ( -- )
11
12 : BFLN ( -- )
13 LNINS PBLL ( TAKE LINE )
14 BOL LMOVE ( UP FROM )
15 CSHOW <BFROT ; ( BUFFER ) ==>

```

Screen: 81

```

0 ( ValFORTH Video editor V1.1 )
1
2 : ARROW ( -- )
3 CBLANK
4 88 @ 541 + DUP @
5 COND
6 3341 = << 30 7453 >>
7 7453 = << 00 0000 >>
8 NOCOND
9 30 3341
10 CONDEND
11 3 PICK !
12 SWAP 2+ C!
13 1 3 UPSTAT !
14 CSHOW ;
15 -->

```

Screen: 79

```

0 ( ValFORTH Video editor V1.1 )
1
2 : BFRPL ( -- )
3 CBLANK
4 !SCR 4 LNFLG ! ( TAKE LINE )
5 PBLL BOL LMOVE ( UP TO SCR )
6 <BFROT CSHOW ( & ROTATE )
7 EDMRK ;
8
9 : TAB ( -- )
10 CBLANK XLOC @ DUP
11 31 = IF DROP -1 ENDIF
12 4 + 4 / 4 * DUP 30 >
13 IF DROP 31 ENDIF
14 XLOC ! CSHOW ;
15 -->

```

Screen: 82

```

0 ( ValFORTH Video editor V1.1 )
1
2 : OOPS ( -- )
3 LNFLG @
4 IF
5 CBLANK
6 PAD 88 @ 32 + 512 >BSCD
7 CSHOW
8 0 LNFLG !
9 ENDIF ;
10
11
12
13
14
15 ==>

```

Screen: 80

```

0 ( ValFORTH Video editor V1.1 )
1
2 : RUB ( -- )
3 XLOC @ 0= NOT ( ON L-EDGE? )
4 IF
5 LFCUR ( RUB IF NOT )
6 0 CURLOC C!
7 CSHOW EDMRK
8 ENDIF
9 INSRT @
10 IF
11 BYTDEL
12 ENDIF ;
13
14
15 ==>

```

Screen: 83

```

0 ( ValFORTH Video editor V1.1 )
1
2 : SPLIT ( -- )
3 YLOC @ 15 <>
4 IF
5 CBLANK
6 LNINS
7 BOL DUP 32 + SWAP
8 XLOC @ CMOVE
9 BOL 32 +
10 XLOC @ ERASE
11 CSHOW
12 ENDIF ;
13
14
15 -->

```


Screen: 84

```

0 ( ValFORTH Video editor V1.1 )
1
2 : SCRSV ( -- )
3 88 @ 32 + PAD 512 BSCD)
4 4 @
5 DO
6 I UPSTAT @
7 @ I UPSTAT !
8 IF
9 PAD 128 I * +
10 TBLK @ I + BLOCK
11 128 CMOVE UPDATE
12 ENDIF
13 LOOP
14 @ INSRT !
15 @ XLOC ! @ YLOC ! ; ==>

```

Screen: 87

```

0 ( ValFORTH Video editor V1.1 )
1
2 : PRVSCR -1 NWSCR ; ( -- )
3
4 : NXTSCR 1 NWSCR ; ( -- )
5
6 : SPLCHR 1 ?ESC ! ; ( -- )
7
8 : EXIT ( -- )
9 HMCUR 19 LSTCHR ! ;
10
11 : EDTABT ( -- )
12 @ UPSTAT @ ERASE
13 EXIT ;
14
15 -->

```

Screen: 85

```

0 ( ValFORTH Video editor V1.1 )
1
2 : SCRGT ( -- )
3 4 @
4 DO
5 TBLK @
6 I + BLOCK
7 PAD 128 I * +
8 128 CMOVE
9 LOOP
10 PAD 88 @ 32 +
11 512 >BSCD ;
12
13
14
15 -->

```

Screen: 88

```

0 ( ValFORTH Video editor V1.1 )
1
2 : PTCHR ( -- )
3 INSRT @ EDMRK
4 IF BYTINS ENDIF
5 LSTCHR @ 127 AND
6 DUP LSTCHR !
7 >SCD CURLOC C!
8 RTCUR XLOC @ @=
9 IF DNCUR ENDIF
10 @ ?ESC ! CSHOW ;
11
12 : CONTROL ( n -- )
13 SEL 19 -> EXIT 17 -> EDTABT
14 28 -> UPCUR 29 -> DNCUR
15 ==>

```

Screen: 86

```

0 ( ValFORTH Video editor V1.1 )
1
2 : NWSCR ( -1/0/1 -- )
3 CBLANK DUP
4 IF SCRSV ENDIF 2* 2*
5 TBLK @ + @ MAX TBLK ! SCRGT
6 TBLK @ @ /MOD
7 DUP <ROT SCR !
8 IF 44 ELSE 53 ENDIF
9 ?1K NOT
10 IF
11 44 = SWAP 2* + DUP SCR ! @
12 ENDIF
13 88 @ 17 + C!
14 @ 84 C! 11 85 ! 1 752 C!
15 . 2 SPACES CSHOW ; ==>

```

Screen: 89

```

0 ( ValFORTH Video editor V1.1 )
1
2 30 -> LFCUR 31 -> RTCUR
3 126 -> RUB 127 -> TAB
4 9 -> INTGL 155 -> NXTLN
5 255 -> BYTINS 254 -> BYTDEL
6 157 -> LNINS 156 -> LNDEL
7 18 -> BFROT 2 -> <BFROT
8 3 -> BFCLR 11 -> >BFNXT
9 20 -> >BFLN 6 -> BFLN>
10 16 -> PRVSCR 14 -> NXTSCR
11 27 -> SPLCHR 8 -> CLREOL
12 1 -> ARROW 21 -> BFRPL
13 15 -> OOPS 10 -> SPLIT
14 NOSEL PTCHR
15 SELEND ; -->

```

Screen: 90

```

0 ( ValFORTH Video editor V1.1 )
1
2 : (V) ( TBLK -- )
3 DECIMAL
4 DUP BLOCK DROP TBLK !
5 1 PFLAG ! 0 GR. 1 752 C! CLS
6 1 559 C@ 252 AND OR 559 C!
7 112 560 @ 6 + C!
8 112 560 @ 23 + C!
9 ." Screen #" 11 SPACES
10 ." #Bufs: " BLEN 32 / . HIDCHR
11 0 UPSTAT 8 ERASE 0 NWSCR
12 PAD ?PADSM @ OVER ?PADSM ! =
13 PBL @ ?BUFSM @ = AND NOT
14 IF PBL BLEN ERASE ENDIF
15 ==>

```

Screen: 91

```

0 ( ValFORTH Video editor V1.1 )
1 BFSHW
2 BEGIN
3 INKEY$ DUP LSTCHR ! -DUP
4 IF
5 ?ESC @
6 IF DROP PTCHR @ LSTCHR !
7 ELSE CONTROL ENDIF
8 ELSE
9 INSRT @
10 IF
11 CBLANK CSHOW
12 ENDIF
13 ENDIF
14 LSTCHR @ 19 =
15 UNTIL -->

```

Screen: 92

```

0 ( ValFORTH Video editor V1.1 )
1
2 CBLANK SCRSV 0 767 C!
3 2 560 @ 6 + C!
4 2 560 @ 23 + C!
5 PBL @ ?BUFSM !
6 2 559 C@ 252 AND OR 559 C!
7 0 LNFLAG ! 0 752 C! CLS CR
8 ." Last edit on screen # "
9 SCR @ . CR CR @ INSRT ! ;
10
11 FORTH DEFINITIONS
12
13 : V ( s -- )
14 1 MAX B/SCR *
15 EDITOR (V) ; ==>

```

Screen: 93

```

0 ( ValFORTH Video editor V1.1 )
1
2 : L ( -- )
3 SCR @ DUP 1+
4 B/SCR * SWAP B/SCR *
5 EDITOR TBLK @ DUP <ROT
6 <= <ROT > AND
7 IF
8 EDITOR TBLK @
9 ELSE
10 SCR @ B/SCR *
11 ENDIF
12 EDITOR (V) ;
13
14
15 -->

```

Screen: 94

```

0 ( ValFORTH Video editor V1.1 )
1
2 : CLEAR ( s -- )
3 B/SCR * B/SCR 0+S
4 DO
5 FORTH I BLOCK
6 B/BUF BLANKS UPDATE
7 LOOP ;
8
9 : COPY ( s1 s2 -- )
10 B/SCR * OFFSET @ +
11 SWAP B/SCR * B/SCR 0+S
12 DO DUP FORTH I
13 BLOCK 2- !
14 1+ UPDATE
15 LOOP DROP ( FLUSH ) ; ==>

```

Screen: 95

```

0 ( ValFORTH Video editor V1.1 )
1
2 : CLEARS ( s # -- )
3 OVER >R 0+S
4 2DUP CR
5 ." Clear from SCR " . CR
6 ." thru SCR " 1 - . Y/N
7 IF
8 DO
9 FORTH I CLEAR
10 LOOP
11 ELSE
12 2DROP
13 ENDIF
14 R) SCR ! FLUSH ;
15 -->

```

Screen: 96

```
0 ( ValFORTH Video editor V1.1 )
1
2 : WHERE EDITOR      ( n n --- )
3   OVER OVER
4   DUP 65532 AND
5   SWAP OVER - 128 *
6   ROT + 32 /MOD
7   YLOC C!
8   2- 0 MAX XLOC C!
9   1 INSRT !
10  EDITOR (V) ;
11
12 : #BUFS              ( # -- )
13   5 MAX 320 MIN 32 * EDITOR
14   ' BLEN ! 0 ?PADSM ! ;
15                               ==>
```

Screen: 99

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 97

```
0 ( ValFORTH Video editor V1.1 )
1
2 : (LOC)              ( sys )
3   BLK @ , IN @ C, ;
4
5 : LOCATOR            ( f -- )
6   IF
7     [ ' (LOC) CFA ] LITERAL
8   ELSE
9     [ ' NOOP CFA ] LITERAL
10  ENDIF
11  ' CREATE ! ;
12
13
14
15                               -->
```

Screen: 100

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 98

```
0 ( ValFORTH Video editor V1.1 )
1
2 : LOCATE
3   [COMPILE] ' DUP NFA 1- DUP
4   2- @ DUP 1439 U< SWAP 0# AND
5   IF
6     SWAP DROP DUP C@
7     SWAP 2- @ WHERE 2DROP
8   ELSE
9     CR ." Cannot locate"
10    ' ( DCMPR DROP DCMPR
11    )( 2DROP CR )
12  ENDIF ;
13
14
15                               BASE !
```

Screen: 101

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 102

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 105

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 103

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 106

0 (Hi-resolution text printing)
1
2 BASE @ DCX
3
4 ' (>SCD) (26 KLOAD)
5 ' (COND) (28 KLOAD)
6
7 57344 VARIABLE GCBAS
8 0 VARIABLE GCPtr
9 2 VARIABLE GCLFT
10 39 VARIABLE GCRGT
11 0 VARIABLE GMOD
12 0 VARIABLE GCCOL
13 0 VARIABLE GCROW
14 120 VARIABLE VMI#
15 ==>

Screen: 104

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 107

0 (Hi-res: GCR)
1
2 : GCR (--)
3 1 GCROW @ + DUP 20
4 703 C@ MAX <
5 IF GCROW !
6 ELSE
7 DROP 88 @ 320 0+S
8 703 C@ 4 =
9 IF 6400 ELSE 7680 ENDIF 2DUP
10 + 320 - >R CMOVE
11 R> 320 ERASE
12 ENDIF
13 GCROW @ 320 *
14 GCLFT @ DUP GCCOL !
15 + GCPtr ! ; -->

Screen: 108

```

0 ( Hi-res: [GCEMIT] )
1
2 : (GCEMIT) ( c -- )
3 >SCD 8 * GCBAS @ +
4 GCPTR @ 88 @ + 320 0+S
5 DO
6     DUP C@ GMOD C@
7     IF I C@ OR ENDIF
8     I C! 1+
9     40 /LOOP
10 DROP 1 GCPTR +!
11 1 GCCOL @ + DUP GCRGT @ )
12 IF DROP GCR
13 ELSE GCCOL !
14 ENDIF ;
15 ==>

```

Screen: 109

```

0 ( Hi-res: GCBKS OSTRIKE GCINIT)
1
2 : GCBKS ( -- )
3 GCCOL @ GCLFT @ )
4 IF
5     -1 GCCOL +! ( backspace )
6     -1 GCPTR +!
7 ENDIF ;
8
9 : OSTRIKE ( f -- )
10 GMOD ! ; ( overstrike)
11
12 : GCINIT ( -- )
13 0 GCROW ! GCLFT @ DUP
14 GCCOL ! GCPTR ! ;
15 -->

```

Screen: 110

```

0 ( Hi-res: GCPOS SUPER SUB )
1
2 : GCPOS ( col row -- )
3 2DUP 320 * + GCPTR !
4 GCROW ! GCCOL ! ;
5
6 : SUPER ( -- )
7 VMI# @ MINUS GCPTR +! ;
8
9 : SUB ( -- )
10 VMI# @ GCPTR +! ;
11
12
13
14
15 ==>

```

Screen: 111

```

0 ( Hi-res: GCEMIT GCTYPE )
1
2 : GCEMIT ( chr -- )
3 DUP
4 COND
5     28 = (( DROP SUPER ))
6     29 = (( DROP SUB ))
7     30 = (( DROP GCBKS ))
8 NOCOND (GCEMIT)
9 CONDEND ;
10
11 : GCTYPE ( adr count -- )
12 0 MAX -DUP
13 IF 0+S DO I C@ GCEMIT LOOP
14 ELSE DROP
15 ENDIF ; -->

```

Screen: 112

```

0 ( Hi-res: [GC"] GC" )
1
2 : (GC") ( -- )
3 R COUNT DUP 1+ R) + >R
4 GCTYPE ;
5
6
7 : GC" ( -- )
8 34 STATE @
9 IF
10     COMPILE (GC") WORD
11     HERE C@ 1+ ALLOT
12 ELSE
13     WORD HERE COUNT GCTYPE
14 ENDIF ; IMMEDIATE
15 ==>

```

Screen: 113

```

0 ( Hi-res: GCSPACE[S] GCD.R )
1
2 : GCSPACE ( -- )
3 BL GCEMIT ;
4
5 : GCSPACES ( n -- )
6 0 MAX -DUP
7 IF 0
8     DO GCSPACE
9     LOOP
10 ENDIF ;
11
12 : GCD.R ( d n -- )
13 >R SWAP OVER DABS
14 (< #S SIGN #> R) OVER -
15 GCSPACES GCTYPE ; -->

```

Screen: 114

```
0 ( Hi-res: GC.R GC. GCLEN )
1
2 : GC.R ( n n -- )
3 >R S->D R> GCD.R ;
4
5 : GC. ( n -- )
6 0 GC.R GCSPACE ;
7
8 : GCLEN ( adr cnt -- #chrs )
9 0 (ROT 0+S
10 DO I C@ 28 -
11 CASE 0 0 0
12 NOCASE 1
13 CASEEND +
14 LOOP ;
15 ==>
```

Screen: 117

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 115

```
0 ( Hi-res: VMI GC.$ )
1
2 : VMI ( n -- )
3 40 * VMI# ! ;
4
5 : GC$. ( adr -- )
6 COUNT GCTYPE ;
7
8 : GCLS ( -- )
9 88 @
10 703 C@ 4 =
11 IF 6400 ELSE 7680 ENDIF
12 ERASE
13 GCRGT @ 0 GCPOS ;
14
15 GCINIT BASE !
```

Screen: 118

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 116

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 119

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 120

```

0 ( Double: DVAR DCON D- D>R DR)
1 BASE @ DCX
2
3 : DARIABLE      ( cccc -- adr )
4 VARIABLE , ;
5
6 : DCONSTANT      ( cccc -- d )
7 (BUILDS , ,
8 DOES> D@ ;
9
10 0. DCONSTANT 0. 1. DCONSTANT 1.
11
12 : D-              ( d d -- d )
13 DMINUS D+ ;
14
15                      ==>

```

Screen: 121

```

0 ( Double: D@= D= D@< D< D> )
1
2 : D@=              ( d -- f )
3 OR @= ;
4
5 : D=                ( d d -- f )
6 D- D@= ;
7
8 : D@<               ( d -- f )
9 SWAP DROP @< ;
10
11 : D<                ( d d -- f )
12 D- D@< ;
13
14 : D>                ( d d -- f )
15 2SWAP D< ;      -->

```

Screen: 122

```

0 ( Double: DMIN DMAX )
1
2 : DMIN              ( d d -- d )
3 2OVER 2OVER D>
4 IF
5 2SWAP
6 ENDIF
7 2DROP ;
8
9 : DMAX              ( d d -- d )
10 2OVER 2OVER D<
11 IF
12 2SWAP
13 ENDIF
14 2DROP ;
15                      ==>

```

Screen: 123

```

0 ( Double: D>R DR> D, M+ )
1
2 : D>R              ( d -- )
3 R> (ROT SWAP )R>R>R ;
4
5 : DR>              ( -- d )
6 R> R> R> SWAP ROT )R ;
7
8 : D,                ( d -- )
9 , , ;
10
11 : M+                ( d n -- d )
12 S->D D+ ;
13
14
15                      -->

```

Screen: 124

```

0 ( Double: DU< )
1
2 : DU<
3 DUP 4 PICK XOR @<
4 IF
5 2DROP D@< NOT
6 ELSE
7 D- D@<
8 ENDIF ;
9
10
11
12
13
14
15                      BASE !

```

Screen: 125

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 126

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 129

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 127

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 130

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 128

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 131

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 132

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 135

0 (Utils:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

-->

Screen: 133

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 136

0 (Utils: XR/W)
1
2 : XR/W (#secs a blk# f --)
3 4 PICK 0
4 DO
5 3 PICK I B/BUF * +
6 3 PICK I + 3 PICK R/W
7 LOOP
8 2DROP 2DROP ;
9
10
11
12
13
14
15

==>

Screen: 134

0 (Utils: Initialization)
1
2 BASE @ DCX
3
4 '(XC!)(21 KLOAD)
5 '(HIDCHR)(24 KLOAD)
6 '(>BSCD)(26 KLOAD)
7
8
9
10
11
12
13
14
15

==>

Screen: 137

0 (Utils: SMOVE)
1
2 : SMOVE (org des cnt --)
3 FLUSH MTB
4 741 @ PAD DUP 1 AND -- 2DUP
5 SWAP B/SCR * B/BUF * U<
6 IF CR ." Too many: "
7 B/BUF B/SCR * / U.
8 ." max." DROP 2DROP
9 ELSE DROP
10 >R DCX MTB CR
11 ." SMOVE from " OVER DUP 3 .R
12 ." thru " R + 1- 3 .R CR
13 8 SPACES
14 ." to " DUP DUP 3 .R
15 ." thru " R + 1- 3 .R -->

Screen: 138

```

0 ( Utils:  SMOVE          )
1
2   SPACE Y/N
3   IF
4   CR ." Insert source" RETURN
5   R B/SCR * PAD DUP 1 AND -
6   4 ROLL B/SCR * OFFSET @ +
7   1 XR/W
8   CR ." Insert dest." RETURN
9   R B/SCR * PAD DUP 1 AND -
10  ROT B/SCR * OFFSET @ +
11  0 XR/W
12  ELSE R) DROP 2DROP
13  CR ." Smove aborted..." CR
14  ENDIF
15  ENDIF ;                      ==>

```

Screen: 139

```

0 ( Utils:  LOADS  THRU          )
1
2
3 : LOADS                      ( n cnt -- )
4  O+S
5  DO
6  I LOAD ?EXIT
7  LOOP ;
8
9
10 : THRU                      ( n n -- n cnt )
11  OVER - 1+ ;
12
13
14
15                               -->

```

Screen: 140

```

0 ( Utils:  SEC  MSEC          )
1
2 : SEC                        ( n -- )
3  0 DO
4  9300 0
5  DO
6  LOOP
7  LOOP ;
8
9 : MSEC                        ( n -- )
10 0 DO
11 6 0
12 DO
13 LOOP NOOP
14 LOOP ;
15                               ==>

```

Screen: 141

```

0 ( Utils:  H->L  L->H  H/L          )
1
2 HEX
3
4 CODE H->L                    ( n -- n )
5   B5 C, 01 C, 95 C, 00 C,
6   94 C, 01 C, 4C C, NEXT , C;
7
8 CODE L->H                    ( n -- n )
9   B5 C, 00 C, 95 C, 01 C,
10  94 C, 00 C, 4C C, NEXT , C;
11
12 CODE H/L                    ( n -- n n )
13   B5 C, 00 C, 94 C, 00 C,
14   4C C, PUSH0A , C;
15 DCX                          -->

```

Screen: 142

```

0 ( Utils:  BIT  ?BIT  TBIT          )
1 HEX
2 CODE BIT                    ( b -- n )
3   B4 C, 00 C, C8 C, A9 C, 00 C,
4   95 C, 00 C, 95 C, 01 C, 38 C,
5   36 C, 00 C, 36 C, 01 C, 18 C,
6   88 C, D0 C, F8 C, 4C C, NEXT ,
7 C;
8 : ?BIT BIT AND 0# ; ( n b -- f )
9
10 : TBIT BIT XOR ; ( n b -- n )
11
12 : SBIT BIT OR ; ( n b -- n )
13
14 : RBIT                    ( n b -- n )
15  FFFF SWAP TBIT AND ;      ==>

```

Screen: 143

```

0 ( Utils:  STICK          )
1 HEX
2 HERE DUP 2DUP 0 , 1 , -1 , 0 ,
3
4 CODE STICK                    ( n -- h v )
5
6   B4 C, 00 C, B9 C, 78 C, 02 C,
7   48 C, CA C, CA C, 29 C, 03 C,
8   0A C, A8 C, B9 C, , 95 C,
9   02 C, C8 C, B9 C, , 95 C,
10  03 C, 68 C, 4A C, 4A C, 29 C,
11  03 C, 0A C, A8 C, B9 C, ,
12  95 C, 00 C, C8 C, B9 C, ,
13  95 C, 01 C, 4C C, ' SWAP ,
14
15 CURRENT @ CONTEXT !      -->

```

Screen: 144

```

0 ( Utils:  STRIG  PADDLE      )
1 HEX
2
3
4 CODE PADDLE      ( n -- n )
5   B4 C, 00 C,  B9 C, 270 ,
6   4C C, PUT0A , C;
7
8 CODE STRIG      ( n -- f )
9   B4 C, 00 C,  B9 C, 284 ,
10  49 C, 01 C,  4C C, PUT0A , C;
11
12 CODE PTRIG      ( n -- f )
13  B4 C, 00 C,  B9 C, 27C ,
14  49 C, 01 C,  4C C, PUT0A , C;
15                               ==>

```

Screen: 145

```

0 ( Utils:  ATTRACT  NXTATR      )
1
2 DCX
3
4 : ATTRACT      ( f -- )
5   IF 255 ELSE 0 ENDIF 77 C! ;
6
7 : NXTATR
8   255 20 C! ;      ( -- )
9 ( Changes user clock )
10
11 : HLDATR
12  0 20 C! ;      ( -- )
13 ( Changes user clock )
14
15                               -->

```

Screen: 146

```

0 ( Utils:  16TIME      )
1 HEX
2
3 CODE 16TIME
4   CA C, CA C,
5   A5 C, 13 C,  95 C, 01 C,
6   A5 C, 14 C,  95 C, 00 C,
7   D0 C, 04 C,
8   A5 C, 13 C,  95 C, 01 C,
9   4C C, NEXT , C;
10
11
12
13
14
15                               ==>

```

Screen: 147

```

0 ( Utils:  BRND  16RND  CHOOSE  )
1 HEX
2
3 CODE BRND      ( -- b )
4   AD C, D20A ,
5   4C C, PUSH0A ,
6   C;
7
8 CODE 16RND      ( -- n )
9   AD C, D20A , 48 C, 68 C, 48 C,
10  68 C, 48 C, AD C, D20A ,
11  4C C, PUSH , C;
12
13 : CHOOSE      ( n -- n )
14  16RND U* SWAP DROP ;
15                               -->

```

Screen: 148

```

0 ( Utils:  CSHUFL  SHUFL      )
1 DCX
2 : CSHUFL      ( a n -- )
3   1- 0 SWAP
4   DO
5     DUP I CHOOSE + OVER I +
6     2DUP C@ SWAP C@
7     ROT C! SWAP C!
8     -1 +LOOP DROP ;
9
10 : SHUFL      ( a n -- )
11  1- 0 SWAP
12  DO DUP I CHOOSE 2* +
13    OVER I 2* +
14    2DUP @ SWAP @ ROT ! SWAP !
15    -1 +LOOP DROP ;      ==>

```

Screen: 149

```

0 ( Utils:  H.  A.      )
1
2 : A.      ( a -- )
3   C@ 127 AND
4   DUP 32 ( OVER
5   124 ) OR
6   IF DROP 46 ENDIF
7   SPEMIT ;
8
9 ' ( H. -- ) ( )
10
11 : H.      ( d -- )
12  BASE @ HEX SWAP
13  0 ( # # # # ) TYPE
14  BASE ! ;
15                               -->

```

Screen: 150

```
0 ( Utils: DUMP )
1 DCX
2
3 : DUMP ( a n -- )
4 O+S
5 DO
6 CR I H->L H. I H.
7 2 SPACES I 8 O+S 2DUP
8 DO
9 I C@ H. SPACE
10 LOOP CR 7 SPACES
11 DO
12 I A. 2 SPACES
13 LOOP ?EXIT
14 8 /LOOP
15 CR ; ==>
```

Screen: 153

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 151

```
0 ( Utils: BLKOP -- system )
1 HEX
2
3 CODE BLKOP ( adr cnt byte -- )
4 A9 C, 03 C, 20 C, SETUP ,
5 HERE C4 C, C4 C, D0 C,
6 07 C, C6 C, C5 C, 10 C, 03 C,
7 4C C, NEXT , B1 C, C6 C,
8 A5 C, C2 C, 91 C, C6 C, C8 C,
9 D0 C, EC C, E6 C, C7 C, 4C C,
10 , DCX
11 C;
12
13
14
15 -->
```

Screen: 154

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 152

```
0 ( Utils: BXOR )
1 HEX
2 CODE BXOR ( adr cnt byte -- )
3 A9 C, 45 C,
4 8D C, ' BLKOP 12 + ,
5 4C C, ' BLKOP , C;
6
7 CODE BAND ( adr cnt byte -- )
8 A9 C, 25 C,
9 8D C, ' BLKOP 12 + ,
10 4C C, ' BLKOP , C;
11
12 CODE BOR ( adr cnt byte -- )
13 A9 C, 05 C,
14 8D C, ' BLKOP 12 + ,
15 4C C, ' BLKOP , C; BASE !
```

Screen: 155

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 156

```

0 ( Strings: -TEXT
1 BASE @ DCX
2 : -TEXT ( a u a -- )
3 2DUP + SWAP
4 DO
5 DROP 1+
6 DUP 1- C@
7 I C@ - DUP
8 IF
9 DUP ABS
10 / LEAVE
11 ENDIF
12 LOOP
13 SWAP DROP DUP
14 IF 1 SWAP +- ENDIF ;
15 ==>

```

Screen: 159

```

0 ( Strings: $CON , $VAR , [ " ] )
1
2 : $CONSTANT ( $ ccc -- )
3 PAD 512 + SWAP OVER $!
4 0 VARIABLE -2 ALLOT
5 HERE $! HERE C@ 1+ ALLOT ;
6
7 : $VARIABLE ( len ccc -- )
8 0 VARIABLE
9 1- ALLOT ;
10
11 : ( " ) ( -- $ )
12 R DUP C@ 1+ R) + >R ;
13
14
15 -->

```

Screen: 157

```

0 ( Strings: -NUMBER
1
2 0 VARIABLE NFLG
3
4 : -NUMBER ( addr -- d )
5 BEGIN DUP C@ BL = DUP + NOT
6 UNTIL 0 NFLG ! 0 0 ROT DUP 1+
7 C@ 45 = DUP >R + -1
8 BEGIN DPL ! (NUMBER) DUP C@
9 DUP BL < > SWAP 0# AND
10 WHILE DUP C@ 46 - NFLG !
11 0 REPEAT DROP R) IF DMINUS
12 ENDIF NFLG @
13 IF 2DROP 0 0 ENDIF
14 NFLG @ NOT NFLG ! ;
15 -->

```

Screen: 160

```

0 ( Strings: "
1
2 : "
3 34 ( Ascii quote )
4 STATE @
5 IF ( cccc" -- )
6 COMPILE ( " ) WORD
7 HERE C@ 1+ ALLOT
8 ELSE
9 WORD HERE ( cccc" -- $ )
10 PAD $! PAD
11 ENDIF ;
12
13 IMMEDIATE
14
15 ==>

```

Screen: 158

```

0 ( Strings: UMOVE , $!
1
2
3 FORTH DEFINITIONS
4
5 : UMOVE ( a a n -- )
6 (ROT OVER OVER U(
7 IF
8 ROT CMOVE
9 ELSE
10 ROT CMOVE
11 ENDIF ;
12
13 : $!
14 OVER C@ 1+ UMOVE ;
15 ==>

```

Screen: 161

```

0 ( Strings: $. , $XCHG
1
2 : $. ( $ -- )
3 DUP C@ 0)
4 IF
5 COUNT TYPE
6 ELSE
7 DROP
8 ENDIF ;
9
10
11 : $XCHG ( $1 $2 -- )
12 DUP PAD 256 + $!
13 OVER SWAP $!
14 PAD 256 + SWAP $! ;
15 -->

```

Screen: 162

```

0 ( Strings: $+ , LEFT$ )
1
2 : $+ ( $1 $2 -- $ )
3 SWAP PAD 256 +
4 >R R $!
5 DUP C@ SWAP 1+
6 R C@ 1+ R +
7 3 PICK UMOVE
8 R C@ + 255 MIN
9 R C! R) PAD $! PAD ;
10
11 : LEFT$ ( $ N -- $ )
12 SWAP PAD <ROT PAD $!
13 OVER C@ MIN
14 OVER C! ;
15 ==>

```

Screen: 163

```

0 ( Strings: RIGHT$ , MID$ )
1
2 : RIGHT$ ( $ n -- $ )
3 SWAP PAD <ROT PAD $!
4 OVER <ROT OVER C@
5 DUP 4 PICK +
6 <ROT MIN DUP
7 <ROT 1- -
8 SWAP ROT OVER OVER
9 C! 1+ SWAP CMOVE ;
10
11 : MID$ ( $ start len -- $ )
12 3 PICK C@ 1+ ROT -
13 0 MAX ROT SWAP
14 RIGHT$ SWAP OVER
15 C@ MIN OVER C! ; -->

```

Screen: 164

```

0 ( Strings: LEN , ASC , $COMP )
1
2 : LEN ( $ -- length )
3 C@ ;
4
5 : ASC ( $ -- c )
6 1+ C@ ;
7
8 : $COMPARE ( $1 $2 -- f )
9 2DUP C@ SWAP C@ SWAP
10 2DUP MIN <ROT - >R
11 ROT 1+ <ROT SWAP 1+
12 -TEXT -DUP 0=
13 IF R) DUP IF 1 SWAP +- ENDIF
14 ELSE R) DROP ENDIF ;
15 ==>

```

Screen: 165

```

0 ( Strings: $< , $= , $> , SV$ )
1
2 : $< ( $1 $2 -- f )
3 $COMPARE 0< ;
4
5 : $= ( $1 $2 -- f )
6 $COMPARE 0= ;
7
8 : $> ( $1 $2 -- f )
9 $COMPARE 0> ;
10
11 : SAVE$ ( $ -- $ )
12 PAD 512 + SWAP
13 OVER $! ;
14
15 -->

```

Screen: 166

```

0 ( Strings: INSTR )
1
2 0 VARIABLE INCNT
3
4 : INSTR ( $1 $2 -- n )
5 0 INCNT ! 1+ SWAP DUP
6 >R OVER 1- C@ >R 1+
7 DUP 1- C@ R - 1+ 0 MAX
8 OVER + SWAP R) <ROT
9 DO
10 2DUP I -TEXT 0=
11 IF
12 I J - INCNT ! LEAVE
13 ENDIF
14 LOOP
15 2DROP R) DROP INCNT @ ; ==>

```

Screen: 167

```

0 ( Strings: CHR$ , DVAL , VAL )
1
2 : CHR$ ( c -- $ )
3 1 PAD C!
4 PAD 1+ C!
5 PAD ;
6
7 : DVAL ( $ -- d )
8 PAD $! PAD
9 DUP C@ OVER 1+ +
10 0 SWAP C!
11 -NUMBER ;
12
13 : VAL ( $ -- n )
14 DVAL DROP ;
15 -->

```

Screen: 168

```

0 ( Strings: DSTR$ , STR[ING]$ )
1
2 : DSTR$ ( d -- $ )
3 DUP (ROT DABS
4 ( # #S SIGN # )
5 SWAP 1- DUP
6 (ROT C! PAD $! PAD ;
7
8 : STR$ ( d -- $ )
9 S->D DSTR$ ;
10
11 : STRING$ ( n $ -- $ )
12 1+ C@ OVER
13 PAD C! PAD
14 1+ (ROT FILL PAD ;
15 ==>

```

Screen: 171

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 169

```

0 ( Strings: $-TB , #IN$ , IN$ )
1
2 : $-TB ( $ -- $ )
3 DUP DUP 1+ SWAP C@
4 -TRAILING SWAP DROP
5 OVER C! ;
6
7 : #IN$ ( n -- $ )
8 -DUP 0= IF 255 ENDIF
9 PAD 1+ SWAP EXPECT PAD
10 BEGIN 1+ DUP C@ 0= UNTIL
11 PAD 1+ - PAD C! PAD ;
12
13 : IN$ ( -- $ )
14 0 #IN$ ;
15 BASE !

```

Screen: 172

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 170

```

0 CONTENTS OF THIS DISK:
1
2 TRANSIENTS: 36 LOAD
3 ARRAYS & THEIR COUSINS: 42 LOAD
4 KEYSTROKE WORDS: 48 LOAD
5 SCREEN CODE CONVERSION: 52 LOAD
6 CASE STATEMENTS: 56 LOAD
7 valFORTH EDITOR 1.1: 64 LOAD
8 HIGH-RES TEXT: 106 LOAD
9 DOUBLE NUMBER XTENSIONS: 120 LOAD
10 MISCELLANEOUS UTILS: 134 LOAD
11 STRING WORDS: 156 LOAD
12
13
14
15

```

Screen: 173

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages)
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 179

0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

PLAYER-MISSILE GRAPHICS

Software and Documentation
© Copyright 1982
Valpar International

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTHTM
SOFTWARE SYSTEM

PLAYER-MISSILE GRAPHICS

Stephen Maguire

Software and Documentation
© Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

vaIFORTH
T.M.

PLAYER-MISSILE GRAPHICS

Version 1.0
April 1982

The following is a description of commands used in creating seemingly difficult video displays using players and missiles. Used alone or in combination with the other available systems by Valpar International, it is possible to obtain graphic displays which compare with those of the best arcade games. The use of players and missiles (also called "player/missiles") allows the beginner to create high quality moving video displays.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

PLAYER/MISSILE GRAPHICS PACKAGE

XXI. PLAYER/MISSILE GRAPHICS

a)	STROLLING THROUGH PLAYER/MISSILE GRAPHICS	1
b)	PLAYER/MISSILE GLOSSARY	
1)	ENABLING PLAYER/MISSILE GRAPHICS	6
2)	CREATING PLAYERS AND MISSILES	10
3)	MOVING/PLACING PLAYERS AND MISSILES	12
4)	SETTING PLAYER/MISSILE BOUNDARIES	14
5)	COLLISIONS BETWEEN PLAYERS AND MISSILES	16

XXII. CHARACTER EDITOR

User's manual for the character set editor.

XXIII. SOUND EDITOR

Description of the audio-palette sound editor.

XXIV. PLAYER/MISSILE SUPPLIED SOURCE

As knowledge of the internal workings of player/missile graphics is not necessary to use this valFORTH package effectively, the internal workings are not explained in this manual. However, for the serious programmer trying to optimize his/her program in every way, an understanding of these internal workings could at times improve code efficiency and/or speed of execution. For a complete explanation of player/missile graphics at the nut-and-bolt level, see the series of articles by Dave and Sandy Small in Creative Computing.

STROLLING THROUGH PLAYER/MISSILE GRAPHICS

One of the biggest differences between the Atari graphic capabilities and those of most other computers is the Atari's ability to use players and missiles. This discussion will not explain the internal workings of player/missile graphics on the Atari; rather, it will explain how to use the basic commands in this valFORTH package. Before we proceed, please load the player/missile graphic routines from the Player/Missile disk. The directory on screen 170 will show what screen to load. Also, if you have the valFORTH Editor/Utilities package, load in the high speed STICK command found in the Miscellaneous Utilities; otherwise, load in the slower version on your Player/Missile disk. (Check the directory for its location).

To start with, let's get a simple player up on the screen to experiment with. First we must initialize the player/missile graphic system and design the player's image. This is simple:

```

1  PMINIT                ( Initialize for single
                           resolution players )

2  BASE !                ( Change to binary for ease )

LABEL CROSS              ( Give the player image a name )
00011000 C,
00011000 C,
00011000 C,
11111111 C,              ( A large plus sign )
11111111 C,
00011000 C,
00011000 C,
00011000 C,

DECIMAL                  ( Now back into base 10 )

PMCLR                    ( Clear player/missile memory )

ON PLAYERS               ( Turn on the players )

CROSS 8 180 50 0 BLDPLY  ( Build a player )

```

You should now see the cross in the upper right-hand corner of the video screen. Now let's take a look at this and see how it works.

First, players are initialized using the PMINIT command. Players can be in either a single or double resolution mode (double res players are twice as tall). "1 PMINIT" is used for single res players. If we had wanted double res players, we would have used "2 PMINIT".

Next, the player image is created. Since it is much easier to make player images as 1's and 0's, we use binary (base two) number entry. Before we design the image, it must be given a name. The LABEL command does this nicely for us.

Player Missile Graphics 1.0

This image is named CROSS. All that need be done now is to draw the picture. Notice how easy it is to see the image when using base two. Of course, we could have stayed in base 10 and still designed the image, but this is usually more difficult. The word C, after each number simply tells FORTH to store that number in the dictionary. Once the picture is designed, we return to decimal for ease.

Both the PMCLR and ON PLAYERS commands are fairly self-descriptive: PMCLR erases all players and missiles so that no random trash appears when the PLAYERS are turned ON. Next, the BLDPLY (build player) command takes the image named CROSS which is 8 bytes tall and assigns it to player 0 at horizontal location 180 and vertical location 50 on the display. Of course, we could have built player 1, 2, or 3 instead.

The cross should be black. Suppose we wanted a blue or green cross instead. This can be done using the PMCOL (player/missile color) command. Try this:

```
0 9 8 PMCOL      ( player hue lum PMCOL )
```

The cross should now appear blue. This command assigns a BLUE (9) hue with a luminance of 8 to player 0. If the color commands are loaded from the valFORTH disk,

```
0 BLUE 8 PMCOL
```

could have been used with the same results. Try changing the color of the player to GREEN (12) or PINK (4). Note that the default colors for players 2 and 3 make them invisible: Their colors should be set immediately upon being built.

Now that we have a player on the screen, let's move it around. We use the PLYMV (player move) command for this. PLYMV needs to know which player to move (there could be as many as five), how far to move it in the horizontal direction, and how far to move it in the vertical direction. Try this:

```
1 1 0 PLYMV      ( horz vert player PLYMV )
```

This moves player 0 down 1 line and right one horizontal position, thus giving the effect of a diagonal move towards the lower right-hand corner. Try these as well:

```
1 0 0 PLYMV      ( move right one position )
-5 0 0 PLYMV      ( move left five positions )
0 20 0 PLYMV      ( move down 20 lines )
0 -15 0 PLYMV     ( move up 15 lines )
-5 2 0 PLYMV      ( move left five, and down two )
```

That's all there is to moving a player. Positive horizontal offsets move the player right, and negative values move the player left. Likewise, positive vertical offsets move the player down while negative ones move the player up. The following program can be typed in and you will have a joystick controlled player:


```

: JOY
  BEGIN
    0 STICK          ( STICK leaves two offsets )
    0 PLYMV          ( for PLYMV to use. )
    ?TERMINAL
  UNTIL ;

```

JOY <ret>

Move the player with stick 0, the left-most stick port. Press any console button to exit the program.

Currently, if the player is moved off any edge, it "wraps" to the opposite side. In other words, we have an "unbound" player. This is rarely desirable. Normally, we want to restrict player movement to certain boundaries. The PLYMV command has a built in boundary check routine specifically for this reason. Right now, new boundaries are set so wrapping occurs. Let's set some boundaries:

```
60 150 50 200 0 PLYBND
```

This sets the boundaries of player zero to 75 on the left, 150 on the right, 50 on the top, and 200 on the bottom. Type JOY again to verify that you can no longer move freely about the display. Try different boundary settings and experiment to get the feel of the command. Boundary checking can be disabled for any or all of the edges. Setting the left or upper boundary to 0 will disable the check on that edge, likewise, 255 in either the right or lower boundary will do the same.

Let's build another player in the lower right-hand corner of the screen. This time, instead of designing the player ourself, let's borrow the image from the standard Atari character set stored in ROM. The image of the digit zero starts at address 57472. The other numbers follow zero. Try this:

```
57472 16 160 150 1 BLDPLY
```

You should now see the numbers 0 and 1 on your screen. This command builds player 1 with the image at address 57472 that is 16 bytes tall and puts it at horizontal position 160 and vertical position 150. Give this player a color if you want.

Until now, we have been using normal size players. It is possible to make the two players on the display different widths using the PLYWID command. PLYWID expects a width specification of 0 or 2 (normal), 1 (double), or 3 (quadruple). Its command form is:

```
width player PLYWID
```

Thus,

```
3 1 PLYWID
```

should make player one four times its original size. The same can be done with player zero:

```
3 0 PLYWID
```

Player Missile Graphics 1.0

Type JOY again and notice that the width has no effect on movement whatsoever. Also notice that player one is unaffected by movement of player zero.

Now that we have two players on the screen, let's interface both of them to the joystick. Type in the following program:

```
: JOY2
  BEGIN
    0 STICK          ( Record stick movement )
    2DUP             ( Make a copy )
    0 PLYMV          ( Move player 0 )
    SWAP             ( Rotate stick 90 degrees )
    1 PLYMV          ( Move player 1 )
    ?TERMINAL
  UNTIL ;

JOY2 <ret>
```

Notice that when you push the stick up, player zero goes up, but player one moves left. The SWAP instruction exchanges the vertical and horizontal offsets from STICK before moving player one. If we were to take the SWAP out, the players would move identically.

In many applications, it is necessary to know when a player has hit another player or some background image. Fortunately, the Atari computer automatically makes this information available. An entire collection of valFORTH words allows checking of all collisions possible. The most general word is ?COL which simply returns a true flag if anything has hit anything else. Here is an example:

```
: BUMP
  BEGIN
    HITCLR
    0 STICK
    0 PLYMV
    ?COL
    IF
      CR ." oops!"
    ENDIF
    ?TERMINAL
  UNTIL ;

BUMP <ret>
```

Move the player around and watch the results. Every time you hit any letters or player one, the word "oops!" should be printed out. This program is quite simple. First, the HITCLR command is issued which erases any old collision information. If this command were omitted, the first time a collision occurred, "oops!" would be continuously printed out. Next the joystick is read and the player moved. If the player touches anything when moved, the collision registers are set. ?COL reads these registers and leaves a true flag if the player has hit something, and the IF statement will then print out "oops!".

Using other commands found in the glossary, we can tell specifically what the player has hit. For example, the ?PXPF command checks to see if a specific player has hit a playfield, and if so, it returns information indicating which playfield.

Although this discussion was limited to using players, the routines for missiles function similarly and can be found in the following glossary. Two player/missile example programs can be found on your Player/Missile disk. These demonstrate how short player/missile routines can be.

PLAYER/MISSILE GLOSSARY

Enabling Player-Missile Graphics

To make use of players and missiles, the video processor must be activated. Players can be several sizes, they can have different overlap priority schemes, and they can have different colors. The following collection of "words" makes this setup task quite simple. Note: Players and missiles are numbered 0 through 3. The fifth player is numbered as four.

(PMINIT)

(addr res ---)

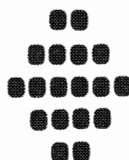
The (PMINIT) command (or PMINIT below) must be used to initialize the player missile routines before any other player missile command may be used. (PMINIT) expects both the address of player/missile memory and a 1 or a 2 indicating whether single or double resolution is desired.

NOTE: The difference between single and double resolution is shown graphically below:

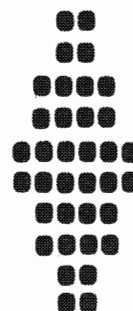
Player as defined
in memory:

```
00011000
00111100
01111110
00111100
00011000
```

single res
on screen:



double res
on screen:



PMINIT

(res ---)

The PMINIT command functions identically to the (PMINIT) command above, except that no address need be given. PMINIT calculates an address based on the current graphic mode. It uses the first unused 2K block of memory below the highest free memory (i.e., below the display list). This should only be used while first learning the system, after that, (PMINIT) should be used to optimize memory utilization. Note that the variable PMBAS contains the calculated address upon return.

PMBAS

(--- addr)

A variable containing the address of player/missile memory. This value must lie on a 2K boundary if single resolution players are used and on a 1K boundary if double resolution players are used. This is set using the (PMINIT) command and is automatically set by the PMINIT command described above. This value should never be set directly, but can be read at any time.

PLAYERS

(ON/OFF ---)

If the flag found on the top of the stack equates to TRUE or ON, then the player/missiles are activated. This does not clear out player missile memory; therefore, the PMCLR command described below is usually used prior to enabling the players and missiles to ensure that no random trash appears on the screen.

If the flag found on the top of the stack equates to FALSE or OFF, then the player/missile graphic mode is de-activated. Turning players off does not clear player-missile memory; therefore, a subsequent ON PLAYERS command would redisplay any previously defined players and missiles. If players are already disabled, the command is ignored.

5THPLY

(flag ---)

In many applications it is desirable to combine the four missiles and simulate a fifth player, thus giving five players (numbered 0-4), and no missiles. If the flag on the stack is non-zero, then the fifth player mode will be initiated; otherwise, the missile mode will be re-activated.

Normally, missiles take on the color of their corresponding players; however, when a fifth player is asked for, all missiles take on the common color of playfield #3. In addition, it also allows the fifth player to be treated exactly as any other player would be treated. Bear in mind that although it is called a "fifth" player, its reference number is four (4). The fifth player is "built" with missile zero on the right, and missile three on the left:

$$|m3|m2|m1|m0| = \text{fifth player}$$

(Note: For convenience, the words ON and OFF have been defined to allow niceties such as:

ON 5THPLY
OFF 5THPLY

These two words are recognized by all words that require an ON/OFF type indication.)

PLYCLR

(pl# ---)

Few applications use all available players. To keep these unused players from displaying trash, they can be cleared of all data by using the PLYCLR command. The PLYCLR command expects the player number on the top of the stack and fills the specified player with zeroes. This command can be used to "turn off" players which are no longer needed.

MSLCLR

(ml# ---)

The MSLCLR command is very much like the PLYCLR command, described above, except that it clears the specified missile. In addition, this can be used when the fifth player is activated to erase parts of the fifth player for special effects.

Player Missile Graphics 1.0

PMCLR

(---)

This command clears all players and all missiles. This is generally used just prior to activating the player-missile graphic mode to ensure that no random trash is placed on the video screen. PMCLR expects no values on the stack, nor does it leave any.

MCPLY

(F ---)

The MCPLY (Multi-Color Player) command expects one value on the top of the stack. If this value is 0 or OFF, then the multi-color player mode is disabled. If this value is 1 or ON, this command instructs the video processor to logically "or" the bits of the colors of player zero with player one, and also of player two with player three. In other words, when players 0 and 1 overlap (or players 2 and 3), a third color (determined by the colors of the overlapping players) will be assigned to the overlapped region rather than assigning one of the players a higher priority. Since players must be one color, this allows for multi-colored players. For example:

Player 0 Pink color (4)	Player 1 Blue color (8)	MCPlayer Pink/blue (4 OR 8 = green)
	BBBB	BBBB
	BBBBBBBB	BBBBBBBB
PPPPPPPP		PPPPPPPP
PPPPPPPP	BB BB	PGGPPGGP
PPPPPPPP		PPPPPPPP
PP PP		PP PP
PPPP		PPPP

NOTE: The lums of the two players are also OR'd.

PRIOR

(n ---)

The PRIOR command expects one value on the top of the stack. This value must be 8, 4, 2, or 1, otherwise unpredictable video displays may occur. PRIOR instructs the video processor as to what has higher priority for a video location on the screen. For example, it will determine whether a plane (a player) will pass in front of a building (a playfield), or whether the plane will pass behind the building. Objects with higher priorities will appear to pass in front of those with lower priorities. The following table shows the available priority settings:

n=8	n=4	n=2	n=1
PFO	PFO	PLO	PLO
PF1	PF1	PL1	PL1
PLO	PF2	PFO	PL2
PL1	PF3*	PF1	PL3
PL2	PLO	PF2	PFO
PL3	PL1	PF3*	PF1
PF2	PL2	PL2	PF2
PF3*	PL3	PL3	PF3*
BAK	BAK	BAK	BAK

* PF3 and PL4 share the same priority

Objects higher on the list will appear to pass in front of objects lower on the list.

CREATING PLAYERS AND MISSILES

Once the player/missile graphics system has been activated and the priorities set, all that need be done is to create the players themselves. Normally, this would be quite difficult to do; however, using the commands and designing techniques described below, this task is made very simple. There are really only three things to do in the creation of a player: setting the width size, setting the color, and creating the picture.

PLYWID (width pl# ---)

The PLYWID command sets the specified player to the desired width. Players are numbered 0, 1, 2, 3, or in the case of the fifth player, 4. Legal widths are:

```
image:      10111101
```

[illegible]

Any other value may cause strange results.

```
MSLWID                                     ( size ml# --- )
```

The MSLWID command is identical to the PLYWID command described above except that it is used to set the size of the missiles. The same size values apply also. The MSLWID command should only be used when in the missile mode (i.e., with the fifth player deactivated).

PMCOL (pl# hue lum ---)

To set the color (hue and lum) of a player, the PMCOL (Player-Missile-Color) command is used. It sets the specified player to the hue and lumina desired. Note that there is no corresponding command to set the colors of missiles as missiles take on the colors of their respective players. To set the color of the 5th player, "pl#" should be 4. If the color words on the valFORTH 1.1 disk are loaded, they can be used to set player colors:

0 BLUE 8 PMCOL

This sets player #0 to a medium blue color.

BLDPLY

(addr len horz vert pl# ---)

The BLDPLY command is probably the most useful of all the commands in this graphic package. It takes an easily predefined picture that resides in memory at address "addr" whose length is "len" and converts it to the specified player "pl#". It then positions the player at the coordinates (horz,vert). The player is then ready to be moved about the screen using the PLYMV command described below.

As an example, a player in the form of an arrow pointing upward will be created, assuming that priorities and such have already been taken care of. Practice has proven that the following method is easiest for creating players:

```

2 BASE !           ( put into binary mode )

LABEL PICTURE      ( the image is named PICTURE )
00011000 C,
00111100 C,
01111110 C,
11011011 C,
00011000 C,
00011000 C,
00011000 C,
00011000 C,
DECIMAL

1 PMINIT           ( initialize for single resolution )
PICTURE 8 80 40 0 BLDPLY

```

Takes the image at location PICTURE which is 8 bytes long, and builds player #0 at location (80,40).

BLDMSL

(addr len horz vert ml# ---)

The BLDPLY command described above does just about everything necessary to create a high-resolution player. The BLDMSL command functions identically to the BLDPLY command except that it is used for setting up missiles (which are in effect just skinny players). The method for creating players can be used for creating missiles as well. Note that if the fifth player mode is activated, the BLDPLY command must be used to create the player.

Building missiles takes a bit more care than building players. Players occupy separate memory, while the four missiles share the same memory. Each missile is two bits wide; all four together are exactly a byte wide. Missile memory is shared with the two lowest bits devoted to missile zero, and the two highest bits devoted to missile three:

```
| m3 | m3 | m2 | m2 | m1 | m1 | m0 | m0 |
```

All players with the same shape can use the same image without any problem since they all are a full byte wide. Missiles, however, cannot use the same shape since their images must be ORed into missile memory. This means that the missile images must be in the proper bit columns. For example, the same image for separate missiles could be:

```

11000000  00110000  00001100  00000011
11000000  00110000  00001100  00000011
11000000  00110000  00001100  00000011
msl#3      msl#2      msl#1      msl#0

```

PUTTING PLAYERS AND MISSILES IN THEIR PLACE

Generally, once a player or missile has been created and put to the video screen, it is moved around. This can be accomplished very easily with the next set of words. Interfacing a movable player with the joystick can improve just about any program which requires input. As a result, it usually gives the program a more professional appearance.

PLYLOC (pl# --- horz vert)

The PLYLOC command (PLaYer LOCation) returns the vertical and horizontal positions of the specified player. This is normally used when a joystick/button setup is being utilized -- i.e., when a joystick is moving a player and the button is used to pinpoint where the player is. A program which draws lines between two dots could use this. The joystick is used to move the player to the desired spot on the screen. Pressing the button tells the program that a selected spot has been made. Once a second spot has been selected, the program then draws a line between them.

MSLLOC (ml# --- horz vert)

The MSLLOC command performs the same function as the PLYLOC command described above except that it is used to find locations of missiles instead of players. Note that using MSLLOC on a fifth player gives meaningless results.

PLYMV (horz vert pl# ---)

The PLaYer MoVe command moves the specified player the direction specified by "vert" and "horz". If "vert" or "horz" is negative, the player is moved up or left respectively, otherwise it is moved down or right unless they happen to be zero in which case nothing happens. The following examples clarify this:

```
0 -5 0 PLYMV ( Move player 0 up 5 lines )
-1 -1 3 PLYMV ( Move player 3 left and up one line )
3 -1 2 PLYMV ( Move player 2 up one dot and right 3 )
```

MSLMV (horz vert ml# ---)

The MSLMV is identical in function as the PLYMV command described above except that it is used to move missiles about the video screen.

PLYPUT (horz vert pl# ---)

The PLYPUT command positions player "pl#" to the location (horz,vert) on the video screen.

PLYCHG

(addr len pl# ---)

Oftentimes it is necessary to change the image of a player after it has been built. The PLYCHG command allows this to be easily done. The PLYCHG command takes the image with length "len" at address "addr" and assigns it to player "pl#". Note that if the new image is shorter than the previous one, part of the previous image will remain. This can be overcome by executing a PLYCLR command prior to PLYCHG.

PLYSEL

(addr # pl# --)

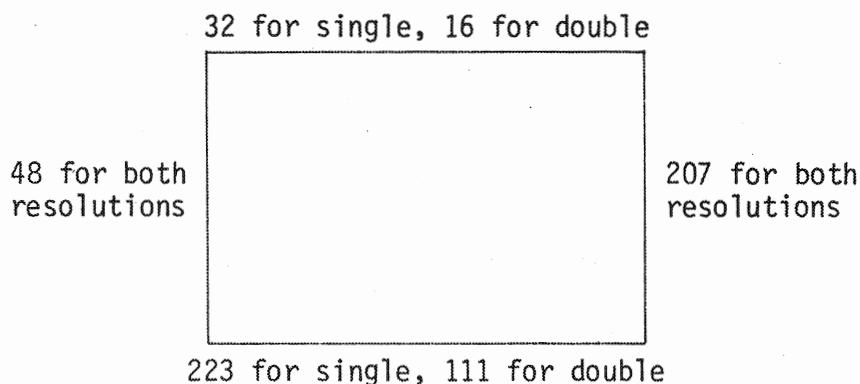
The PLYSEL command is used to select image "#" out of a table of images of the same length and assigns that image to the specified player. PLYSEL is typically used to animate players. An example usage of this can be found in Player/Missile Example #2 found in the directory of the disk.

PLAYER/MISSILE BOUNDARIES

It is often desirable to put limitations on the movements of players and missiles. Boundaries can be set up for each player and missile independently and upon each move command, they will remain within those boundaries. Additionally, a boundary status byte for each player is available for scrutiny at any time. This section explains how this is used.

PLYBND (left right top bottom pl# --)

In most applications, the movements of players are kept within certain boundaries. The PLYBND command frees the user from having to worry about boundary checking. This command expects the player number and all four boundaries. Whenever a PLYMV is then used, the player is always kept within the set boundaries. Also, upon each move a boundary status byte is left in the c-array PLYSTT (see ?PLYSTT below). The edge boundaries of the screen are:



Note that in special cases the boundary checker will fail. If the left boundary is 0 and the player is at the boundary, any move left will not be checked as expected. For example, if it were moved left by one position (-1), the new horizontal position would be -1 or FFFF in hex. Since only 8 bit unsigned comparisons are made, the horizontal position appears to be 255 (FF hex). Post calculating boundary checking turns out to be more useful because it allows any or all edges to be unbounded. If an unbounded player is desired, use this:

0 255 0 255 pl# PLYBND

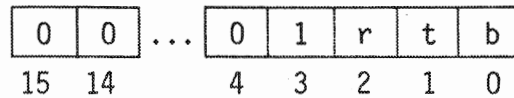
For an example of PLYBND, see the example program found in the directory on screen 170 of your disk.

MSLBND (left right top bottom ml# --)

The MSLBND command is the same as the PLYBND command above, except that it is used for missiles. Upon each move a boundary status byte is left in the array MSLSTT. See ?MSLSTT below.

`?BND``(--- n)`

This command leaves the boundary check status of the last PLYMV or MSLMV performed. The value has the following form:



Only the lower four bits are of use. Each bit represents a different edge. If the bit is set, then the player or missile has attempted to move beyond that boundary. Note that only two of the four bits can be set at any time.

Note: DECIMAL

...

`?BND 3 AND`

`IF hit-vertical-boundary ENDIF`

`?BND 12 AND`

`IF hit-horizontal-boundary ENDIF`

...

`?PLYSTT``(pl# --- val)`

Given a player number, returns the boundary check byte of that player. This byte is the status byte for the most recent PLYMV of that player. See `?BND` above for the description of the status byte.

`?MSLSTT``(ml# --- val)`

Given a missile number, returns the boundary check byte of that missile. This byte is the status byte for the most recent MSLMV of that missile. See `?BND` above for the description of the status byte.

CHECKING FOR INTERACTION BETWEEN PLAYERS

All the commands given so far allow the creation of any player or missile desired. But once that player is on the screen and moving around, it is often necessary to know when two or more objects (players, missiles, and playfields) touch or "crash" into each other. This remaining collection of commands allows checking of all possible "hit" combinations.

?COL (--- f)

The ?COL command is a very general collision detector. It does nothing more than indicate whether two or more objects have "crashed" -- it does not give any indication of what has collided. It leaves a 1 on the stack if a collision has taken place; otherwise it leaves a zero.

?MXPF (m1# --- n)

The ?MXPF command is a much more specific collision detection command. It stands for "?collision of Missile #X with any PlayField". It is used to check if a specific missile has hit any playfield. It returns a zero if no collision has taken place, and leaves an 8, 4, 2, 1, or combinations of these (e.g., 12 = 8+4) if a collision has occurred. Each of these four basic values represents a specific playfield:

3 ?MXPF (Has missile #3 hit any playfields?)

TOS	binary	meaning of val
0	0000	no collisions
1	0001	with pf#0
2	0010	with pf#1
3	0011	with pf#0,1
4	0100	with pf#2
5	0101	with pf#2,0
6	0110	with pf#2,1
7	0111	with pf#2,1,0
8	1000	with pf#3
9	1001	with pf#3,0
10	1010	with pf#3,1
11	1011	with pf#3,1,0
12	1100	with pf#3,2
13	1101	with pf#3,2,0
14	1110	with pf#3,2,1
15	1111	with pf#3,2,1,0

To test for a collision with one specific playfield, use one of the following:

1 AND (Leaves 1 if collision with pf#0, else 0)
2 AND (" 1 " " pf#1, " 0)
4 AND (" 1 " " pf#2, " 0)
8 AND (" 1 " " pf#3, " 0)

?PXPF

(pl# --- n)

The ?PXPF command (?collision of Player #X with any PlayField) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions with players and playfields instead of missiles and playfields.

?MXPL

(ml# --- n)

The ?MXPL command (?collision of Missile #X with any Player) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between missiles and players. Note that it is impossible for a missile to collide with a fifth player since it would be, in effect, colliding with itself.

?PXPL

(pl# --- n)

The ?PXPL command (?collision of Player #X with any other players) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between players. Note that it is impossible for a player to collide with itself.

HITCLR

(---)

The HITCLR command clears all collision registers. In other words, it sets the collision monitor to a state which indicates that no collisions have occurred.

THE CHARACTER SET EDITOR

Character Sets

Whenever the computer has to display a character on the video screen, it must refer to a table which holds the shape definition for that character. By changing this table, new character sets can be formed.

The shape of a single character in the table (or character set) is made up of 8 bytes of data. A character is one byte wide and 8 bytes tall forming an 8 by 8 bit matrix. If a bit in this matrix is set (1), then a dot will appear on the screen. If a bit is reset (0), nothing is displayed. For example, the letter I could be defined as:

.....	00000000	\$00 = 0
.....	01111110	\$7E = 126
.....	00011000	\$18 = 24
.....	00011000	\$18 = 24
.....	00011000	\$18 = 24
.....	00011000	\$18 = 24
.....	01111110	\$7E = 126
.....	00000000	\$00 = 0

Thus, the sequence 0, 126, 24, 24, 24, 24, 126, 0, represents the letter I. The entire alphabet is constructed in this fashion. By selectively setting the bit pattern, custom made characters can be formed. This can find many uses. A British character set can be made by changing the one character "#" to the British monetary symbol. Likewise, a Japanese character set could be made by replacing the lowercase characters with Katakana letters.

Another use would be to design special symbol sets. For example, an entire set could be devoted to special mathematical symbols such as plus-minus signs, square-root signs integration signs, or vector signs. (Although this would be of little use in normal operation where character sets cannot be mixed on the same line, using the high resolution text output routines in the Editor/Utilities package. It becomes easy to mix character sets in this fashion.) Assuming the character sets were defined, it would be possible to have a Japanese quotation (in kana of course) embedded within the text of a mathematical explanation of some kind all on the same line!

A final use for custom character sets is for "map-making." Characters can be designed so that they can be pieced together to form a picture. An excellent example of this can be found in Cris Crawford's Eastern Front game available through the Atari Program Exchange. When done properly, the final "puzzle" will appear as though it is a complicated high resolution picture.

Now, on to the editor...

The Editor

The following description explains how to use the character editor found on the Player/Missile disk. This editor allows a character set to be designed and then saved on disk for later modification or use. A copy of the standard character has already been saved and can be located through the directory on screen 170.

After loading the character editor, it is executed by typing:

CHAR-EDIT <ret>

The screen has an 8 by 8 grid in the upper-lefthand corner. On the right side there is a command list, and at the bottom, a section is reserved to display the current character set.

The Commands:

- I) The joystick
A joystick in port 0 (the leftmost port) is used to move the character cursor (the solid circle) within the 8 by 8 grid. The cursor indicates where the next change to the current character will be made.
- II) The button
When pressed, the joystick button will toggle the bit under the character cursor in the 8 by 8 grid. If the bit is set (on), it will be reset. If the bit is reset (off), it will be set. The character will be updated in the character set found at the bottom of the screen.
- III) "1" command
By pressing the "1" the current character is cleared in both the grid and in the character set at the bottom of the display. There is no verify prompt for this command.
- IV) "2" command
By pressing the "2" key the current character and character set are cleared. User verification is required before any action is taken.
- V) "3" command
By pressing the "3" key the current character is saved to disk. User verification is required with a yes/no response. If a yes response is given, a screen number is asked for and the current character set is saved on the specified screen. The current character is not destroyed upon a save.
- VI) "4" command
By pressing the "4" key a character set is loading from disk, destroying the current character set. User verification is required with a yes/no response. If a yes response is given, a screen number is asked for and a character set loaded from the specified screen.

VII) "<--" and "-->" commands

These two arrow keys move the character pointer through the character set to allow modification of any character in the current set.

VIII) Console key

Pressing any console key terminates the edit session and returns control to the FORTH system. The current character set is lost unless it is saved to disk prior to ending the session.

Loading Character Sets

The following three words allow easy use of custom character sets.

CHLOAD

(addr scr# cnt ---)

The CHLOAD command takes the first "cnt" characters on screen "scr#" and stores them consecutively starting at address "addr". Each screen (in half-K mode) will only hold 64 character definitions. If "cnt" is greater than 64, CHLOAD will continue loading from the next screen. Many character sets could be loaded at one time by giving a very large "cnt" value. Besides being able to load a full set, the CHLOAD command allows the building of a new set from several other sets.

Note that if a 20 character/line mode is being used, "addr" should lie on a half-K boundary (only upper 7 bits significant). If a 40 character/line mode is being used, "addr" should lie on an 1K boundary (only upper 6 bits significant). Also note that PAD is modified by CHLOAD.

SPLCHR

(addr ---)

The SPLCHR commands activates the character set at the address specified.

NMLCHR

(---)

The NMLCHR command re-activates the normal character set.

AUDIO-PALETTE -- A SOUND EDITOR

Audio-Palette is a sound editor which generates all possible time-in dependent sounds that the Atari 400/800 microcomputer can produce. Each of the four channels are interfaced to one of the four joystick ports. The joysticks allow the setting of the pitch (horizontal) the distortion (vertical) of their corresponding channel. When the joystick button is pushed, the sound is made. To get a better idea of how this works, load the editor (see screen 170) and type:

AUDED <ret>

The screen should clear and a table of values should appear at the bottom of the display. In the upper lefthand corner of the screen, there should be four numerals (players) overlayed (one for each channel). Each of these players can be moved around the display by using a joystick in the appropriate port.

As a player is moved vertically, the distortion changes. As a player is moved horizontally, the pitch changes. By pressing the button, a sound will be made according to the current frequency (pitch), distortion, volume, and audio control settings. To increase the volume, the up-arrow is used. Any time the up-arrow is pressed, all channels whose corresponding joystick buttons are pressed will have their volumes increased. Likewise, the down-arrow will decrease the volumes.

Each bit of the audio control value performs some function in the sound generator. The bits are numbered 0 to 7. Pressing the keys 0 to 7 will toggle the corresponding bits in the audio control register. For a description of these bit settings, please refer to the explanation of SOUND in the valFORTH 1.1 package.

XXIV. PLAYER/MISSILE SUPPLIED SOURCE

Screen: 30

```

0 ( PlyMsl: arrays and variables)
1 BASE @
2 DCX '( ARRAY )( 80 KLOAD )
3 0 VARIABLE PMBAS
4 5 CARRAY PLYVRT
5 5 CARRAY PLYHRZ
6 5 CARRAY PLYLEN
7 5 ARRAY PLYADR
8 4 CARRAY MSLVRT
9 4 CARRAY MSLHRZ
10 4 CARRAY MSLLEN
11 4 ARRAY MSLADR
12 5 ARRAY PMADR
13 0 VARIABLE PMLN
14 0 VARIABLE PMRES
15 0 VARIABLE MSLSZ ==>

```

Screen: 31

```

0 ( PlyMsl: arrays and variables)
1
2 0 VARIABLE BOUNDS 34 ALLOT
3 5 CARRAY PLYSTT
4 4 CARRAY MSLSTT
5 0 VARIABLE BNDCOL
6 2 VARIABLE 5THWID
7
8 CTABLE 5THDAT
9 2 C, 4 C, 2 C, 8 C,
10
11 HEX
12
13 CTABLE MSLDAT
14 FC C, F3 C, CF C, 3F C,
15 -->

```

Screen: 32

```

0 ( PlyMsl: [PINIT] )
1
2 : (PINIT) ( addr res -- )
3 SWAP PMBAS ! 1- DUP PMRES !
4 NOT 10 * 0C OR
5 22F C@ EF AND OR 22F C!
6 PMBAS @ 180 PMRES @
7 NOT 1+ >R
8 R * + DUP 4 PMADR !
9 80 R) * >R
10 R + DUP 0 PMADR !
11 R + DUP 1 PMADR !
12 R + DUP 2 PMADR !
13 R + 3 PMADR !
14 R) PMLN ! ;
15 ==>

```

Screen: 33

```

0 ( PlyMsl: PMINIT PLAYERS )
1
2 : PMINIT ( res -- )
3 2E6 C@ 8 - F8 AND
4 OVER 1- 4 * + 100 *
5 SWAP (PMINIT) ;
6
7 : PLAYERS ( f -- )
8 IF
9 PMBAS @ DUP
10 PMRES @ 1+ (PMINIT)
11 SP@ 1+ C@ SWAP
12 DROP D407 C!
13 SGRCTL @ 3 OR DUP
14 SGRCTL ! D01D C!
15 ELSE -->

```

Screen: 34

```

0 ( PlyMsl: 5THPLY )
1
2 SGRCTL @ FC AND
3 DUP SGRCTL ! D01D C!
4 22F C@ E3 AND 22F C!
5 D00D 5 ERASE
6 ENDIF ;
7
8
9 : 5THPLY ( f -- )
10 26F C@ SWAP
11 IF 10 OR
12 ELSE EF AND
13 ENDIF
14 26F C! ;
15 ==>

```

Screen: 35

```

0 ( PlyMsl: PMCLR PLYCLR )
1
2
3 : PMCLR ( -- )
4 4 PMADR @
5 PMLN @ 5 *
6 0 FILL ;
7
8
9 : PLYCLR ( pl# -- )
10 PMADR @
11 PMLN @
12 0 FILL ;
13
14
15 -->

```

Screen: 36

```

0 ( PlyMsl: MSLCLR PRIOR )
1
2 : MSLCLR ( ml# -- )
3 4 PMADR @ DUP
4 PMLen @ + SWAP
5 DO
6 DUP MSLDAT C@
7 I C@ AND I C!
8 LOOP
9 DROP ;
10
11 : PRIOR ( n -- )
12 26F C@ 0F0 AND
13 OR 26F C! ;
14
15 ==>

```

Screen: 37

```

0 ( PlyMsl: PLYMV )
1
2 CODE PLYMV
3 84 C, N 6 + C, B5 C, 00 C,
4 0A C, A8 C, B9 C, 0 PMADR 1+ ,
5 85 C, N 1+ C, B9 C, 0 PMADR ,
6 85 C, N 1- C, B9 C, 0 PLYADR ,
7 85 C, N 2+ C, B9 C, 0 PLYADR
8 1+ , 85 C, N 3 + C, B4 C, 0 C,
9 B9 C, 0 PLYLEN , 85 C, N 4 + C,
10 B9 C, 0 PLYHRZ , 18 C, 75 C,
11 04 C, D9 C, BOUNDS , B0 C, 5 C,
12 B9 C, BOUNDS , E6 C, N 6 + C,
13 06 C, N 6 + C, D9 C, BOUNDS 5 +
14 , F0 C, 07 C, 90 C, 05 C, B9 C,
15 BOUNDS 5 + , E6 C, N 6 + C, -->

```

Screen: 38

```

0 ( PlyMsl: PLYMV )
1 99 C, 0 PLYHRZ , 95 C, 05 C,
2 B9 C, 0 PLYVRT , 85 C, N C,
3 18 C, 75 C, 2 C, 06 C, N 6 + C,
4 D9 C, BOUNDS A + , B0 C, 05 C,
5 B9 C, BOUNDS A + , E6 C, N 6 +
6 C, 6 C, N 6 + C, D9 C, BOUNDS
7 F + , F0 C, 07 C, 90 C, 05 C,
8 B9 C, BOUNDS F + , E6 C, N 6 +
9 C, 99 C, 0 PLYVRT , 95 C, 3 C,
10 38 C, E5 C, N C, B0 C, 05 C,
11 A5 C, N C, 38 C, F5 C, 03 C,
12 95 C, 02 C, C5 C, N 4 + C,
13 90 C, 02 C, A5 C, N 4 + C,
14 85 C, N 5 + C,
15 ==>

```

Screen: 39

```

0 ( PlyMsl: PLYMV )
1 A5 C, N C, D5 C, 03 C, 90 C,
2 08 C, 18 C, 65 C, N 4 + C, 38
3 C, E5 C, N 5 + C, 85 C, N C,
4 18 C, 65 C, N 1- C, 85 C, N C,
5 B5 C, 2 C, F0 C, 0B C, A0 C,
6 00 C, 98 C, 88 C, C8 C, 91 C,
7 N C, C4 C, N 5 + C, D0 C,
8 F9 C, B5 C, 00 C, C9 C, 04 C,
9 D0 C, 14 C, B5 C, 05 C, A0 C,
10 04 C, HERE 88 C, 30 C, 0A C,
11 99 C, D004 , 18 C, 6D C, 5THWID
12 , 4C C, , 4C C, HERE 2 ALLOT
13 B5 C, 05 C, B4 C, 00 C, 99 C,
14 D000 , HERE SWAP ! B4 C, 00 C,
15 A5 C, N 6 + C, -->

```

Screen: 40

```

0 ( PlyMsl: PLYMV )
1
2 99 C, 0 PLYSTT , 8D C, BNDCOL ,
3 B5 C, 3 C, 18 C, 65 C, N 1- C,
4 85 C, N C, A0 C, 00 C,
5 B1 C, N 2+ C,
6 91 C, N C, C8 C, C4 C, N 4 + C,
7 D0 C, F7 C, E8 C, E8 C,
8 4C C, POPTWO , C;
9
10
11
12
13
14
15 ==>

```

Screen: 41

```

0 ( PlyMsl: MSLMV )
1 HEX
2
3 CODE MSLMV
4 84 C, N 6 + C, B5 C, 0 C, 0A C,
5 A8 C, AD C, 4 PMADR 1+ , 85 C,
6 N 1+ C, AD C, 4 PMADR , 85 C,
7 N 1- C, B9 C, 0 MSLADR , 85 C,
8 N 2+ C, B9 C, 0 MSLADR 1+ ,
9 85 C, N 3 + C, B4 C, 0 C, B9 C,
10 0 MSLDAT , 85 C, N 7 + C, B9 C,
11 0 MSLLEN , 85 C, N 4 + C, B9 C,
12 0 MSLHRZ , 18 C, 75 C, 04 C,
13 D9 C, BOUNDS 14 + , B0 C, 5 C,
14 B9 C, BOUNDS 14 + , E6 C, N 6 +
15 -->

```

Screen: 42

```

0 ( PlyMsl: MSLMV )
1
2 C, 6 C, N 6 + C, D9 C, BOUNDS
3 18 + , F0 C, 07 C, 90 C,
4 05 C, B9 C, BOUNDS 18 + ,
5 E6 C, N 6 + C,
6 99 C, 0 MSLHRZ , 95 C, 05 C,
7 B9 C, 0 MSLVRT , 85 C, N C,
8 18 C, 75 C, 02 C, 6 C, N 6 + C,
9 D9 C, BOUNDS 1C + , B0 C, 5 C,
10 B9 C, BOUNDS 1C + , E6 C, N 6 +
11 C, 06 C, N 6 + C, D9 C, BOUNDS
12 20 + , F0 C, 7 C, 90 C, 5 C,
13 B9 C, BOUNDS 20 + , E6 C, N 6 +
14 C, 99 C, 0 MSLVRT , 95 C, 3 C,
15 ==>

```

Screen: 43

```

0 ( PlyMsl: MSLMV )
1
2 38 C, E5 C, N C, B0 C, 5 C, A5
3 C, N C, 38 C, F5 C, 3 C, 95 C,
4 2 C, C5 C, N 4 + C, 90 C, 2 C,
5 A5 C, N 4 + C, 85 C, N 5 + C,
6 A5 C, N C, D5 C, 3 C, 90 C,
7 8 C, 18 C, 65 C, N 4 + C, 38
8 C, E5 C, N 5 + C, 85 C, N C,
9 18 C, 65 C, N 1- C, 85 C, N C,
10 A0 C, FF C, C8 C, B1 C, N C,
11 25 C, N 7 + C, 91 C, N C, C4
12 C, N 5 + C, D0 C, F5 C, B5 C,
13 5 C, B4 C, 0 C, 99 C, D004 ,
14
15 -->

```

Screen: 44

```

0 ( PlyMsl: MSLMV )
1
2 B4 C, 0 C, A5 C, N 6 + C, 99
3 C, 0 MSLSTT , 8D C,
4 BNDCOL , B5 C, 3 C, 18 C,
5 65 C, N 1- C, 85 C, N C,
6 A0 C, 00 C, B1 C, N C,
7 25 C, N 7 + C, 11 C, N 2+ C,
8 91 C, N C, C8 C,
9 C4 C, N 4 + C, D0 C, F3 C, E8
10 C, E8 C, 4C C, POPTWO , C;
11
12
13
14
15 ==>

```

Screen: 45

```

0 ( PlyMsl: BLDPLY BLDMSL )
1
2 : BLDPLY ( a 1 h v pl# -- )
3 >R R PLYVRT C!
4 R PLYHRZ C! R PLYLEN C!
5 R PLYADR ! ( R PLYCLR )
6 0 0 R) PLYMV ;
7
8 : BLDMSL ( a 1 h v pl# -- )
9 >R R MSLVRT C!
10 R MSLHRZ C! R MSLEN C!
11 R MSLADR ! ( R MSLCLR )
12 0 0 R) MSLMV ;
13
14
15 -->

```

Screen: 46

```

0 ( PlyMsl: PLYCHG PLYSEL PLYPUT )
1
2 : PLYCHG ( a len pl# -- )
3 >R R PLYLEN C!
4 R PLYADR !
5 0 0 R) PLYMV ;
6
7 : PLYSEL ( a # pl# -- )
8 >R R PLYLEN C0 * +
9 R PLYLEN C0 R) PLYCHG ;
10
11 : PLYPUT ( h v pl# -- )
12 >R R PLYVRT C0 -
13 SWAP R PLYHRZ C0 -
14 SWAP R) PLYMV ;
15 ==>

```

Screen: 47

```

0 ( PlyMsl: PLYWID )
1
2 CODE PLYWID
3 B5 C, 00 C, C9 C, 04 C, F0 C,
4 09 C, A8 C, B5 C, 02 C, 99 C,
5 D008 , 4C C, HERE 2 ALLOT
6 A8 C, A0 C, 04 C, 0A C, 0A C,
7 15 C, 02 C, 88 C, D0 C, F9 C,
8 8D C, MSLSZ , 8D C, D00C ,
9 B4 C, 02 C, B9 C, 0 5THDAT ,
10 85 C, N C, 8D C, 5THWID ,
11 AD C, 4 PLYHRZ , A0 C, 04 C,
12 HERE 88 C, 30 C, 09 C, 99 C,
13 D004 , 18 C, 65 C, N C, 4C C,
14 , HERE SWAP ! 4C C, POPTWO ,
15 C; -->

```

Screen: 48

```

0 ( PlyMsl: MSLWID )
1
2 CODE MSLWID
3 B4 C, 00 C, B9 C, 0 MSLDAT ,
4 2D C, MSLSZ , HERE
5 88 C, 30 C, 7 C, 16 C, 02 C,
6 16 C, 02 C, 4C C, , 15 C,
7 02 C, 8D C, MSLSZ , 8D C,
8 D00C , 4C C, POPTWO ,
9 C;
10
11
12
13
14
15 ==>

```

Screen: 51

```

0 ( PlyMsl: ?MXPL ?PXPL PLYBND )
1
2 CODE ?MXPL ( ml# -- n )
3 B4 C, 00 C, B9 C, D008 ,
4 4C C, PUT0A , C;
5
6 CODE ?PXPL ( pl# -- n )
7 B4 C, 00 C, B9 C, D00C ,
8 4C C, PUT0A , C;
9
10 CODE HITCLR ( -- )
11 8C C, D01E , 4C C, NEXT , C;
12
13 CODE ?BND ( xl# -- n )
14 AD C, BNDCOL ,
15 4C C, PUSH0A , C; -->

```

Screen: 49

```

0 ( PlyMsl: PLYLOC MSLLOC MCPLY )
1
2 CODE PLYLOC ( pl# -- h v )
3 94 C, 01 C, B4 C, 0 C,
4 B9 C, 0 PLYHRZ , 95 C, 0 C,
5 B9 C, 0 PLYVRT , 4C C, PUSH0A ,
6
7 CODE MSLLOC ( ml# -- h v )
8 94 C, 01 C, B4 C, 0 C,
9 B9 C, 0 MSLHRZ , 95 C, 0 C,
10 B9 C, 0 MSLVRT , 4C C, PUSH0A ,
11
12 : MCPLY ( f -- )
13 26F C@ SWAP
14 IF 20 OR ELSE DF AND ENDIF
15 26F C! ; -->

```

Screen: 52

```

0 ( PlyMsl: MSLBND ?BND )
1
2 CODE ?PLYSTT ( pl# -- n )
3 B4 C, 00 C, B9 C, 0 PLYSTT ,
4 4C C, PUT0A , C;
5
6
7 CODE ?MSLSTT ( ml# -- n )
8 B4 C, 00 C, B9 C, 0 MSLSTT ,
9 4C C, PUT0A , C;
10
11 : PLYBND ( l r t b pl# -- )
12 >R 4 ROLL >R
13 <ROT SWAP R> R>
14 BOUNDS + 14 0+S
15 DO I C! 5 /LOOP ; ==>

```

Screen: 50

```

0 ( PlyMsl: ?COL HITCLR ?MXPF... )
1
2 CODE ?COL ( -- f )
3 CA C, CA C, 98 C, A0 C, 0F C,
4 19 C, D000 , 88 C, 10 C, FA C,
5 C8 C, 94 C, 01 C, 95 C, 00 C,
6 4C C, ' 0# ( CFA @ ) ,
7 C;
8
9 CODE ?MXPF ( ml# -- n )
10 B4 C, 00 C, B9 C, D000 ,
11 4C C, PUT0A , C;
12
13 CODE ?PXP ( pl# -- n )
14 B4 C, 00 C, B9 C, D004 ,
15 4C C, PUT0A , C; ==>

```

Screen: 53

```

0 ( PlyMsl: PMCOL )
1
2 : MSLBND ( l r t b ml# -- )
3 >R 4 ROLL >R
4 <ROT SWAP R> R>
5 BOUNDS + 14 + 10 0+S
6 DO I C! 4 /LOOP ;
7
8 : PMCOL ( pl# col lum -- )
9 SWAP 10 * +
10 SWAP DUP 4 =
11 IF
12 DROP 2C7 C!
13 ELSE
14 2C0 + C!
15 ENDIF ; -->

```

Screen: 54

```
0 ( PlyMsl: initialization )
1
2 DCX
3
4 BOUNDS      36   0 FILL
5 BOUNDS  5 +  5 255 FILL
6 BOUNDS 15 +  5 255 FILL
7 BOUNDS 24 +  4 255 FILL
8 BOUNDS 32 +  4 255 FILL
9
10 0 PLYSTT 5 ERASE
11 0 MSLSTT 4 ERASE
12
13 1 PMINIT      ( Set up defaults )
14
15 BASE !
```

Screen: 57

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 55

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 58

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 56

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 59

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```


Screen: 60

```

0 ( Audio Editor )
1
2 BASE @ DCX
3
4 '( PLYMV )( 15 KLOAD )
5 '( SOUND )( 83 KLOAD )
6 '( STICK )( 84 KLOAD )
7
8
9 VOCABULARY AUDPAL IMMEDIATE
10 AUDPAL DEFINITIONS
11
12 4 CARRAY PIT
13 4 CARRAY VOL
14 4 CARRAY DST
15 0 VARIABLE ACTL ==>

```

Screen: 63

```

0 ( Audio Editor )
1 HEX
2 : SETP ( -- )
3 2 PMINIT PMCLR 1 PRIOR
4 0 3 ( RDORNG ) 6 PMCOL
5 1 8 ( BLUE ) 6 PMCOL
6 2 4 ( PINK ) 8 PMCOL
7 3 1 ( GOLD ) 6 PMCOL
8 4 0
9 DO
10 1 I PLYWID
11 E080 I 8 * + 8 37 15 I
12 BLDPLY
13 LOOP
14 ON PLAYERS ;
15 DCX -->

```

Screen: 61

```

0 ( Audio Editor )
1
2 HEX
3 CTABLE TBL
4 32 C, 1F C, 1E C, 1A C, 18 C,
5 1D C, 1B C, 33 C, 0F C, 0E C,
6 DCX
7
8 : WPIT ( pl# -- )
9 10 OVER 20 + POS. PIT C@
10 3 .R ;
11
12 : WDST ( pl# -- )
13 16 OVER 20 + POS. DST C@
14 2 .R ;
15 -->

```

Screen: 64

```

0 ( Audio Editor )
1
2 : INIT ( -- )
3 0 GR. 1 752 C! CLS 3 19 POS.
4 ." Chan Freq Dist "
5 ." Vol AUDCTL"
6 4 0
7 DO
8 8 I VOL C!
9 0 I PIT C!
10 0 I DST C!
11 CR I 3 SPACES . I WPIT
12 I WDST I WVOL
13 LOOP
14 0 ACTL ! WACTL SETP ;
15 ==>

```

Screen: 62

```

0 ( Audio Editor )
1
2 : WVOL ( pl# -- )
3 20 OVER 20 +
4 POS. VOL C@ 2 .R ;
5
6 : WACTL ( -- )
7 28 21 POS. BASE C@ ACTL C@
8 DUP DUP 3 .R 2 BASE C!
9 26 22 POS. 0
10 (#####) TYPE
11 FILTER! BASE C! ;
12
13
14
15 ==>

```

Screen: 65

```

0 ( Audio Editor )
1
2 : SND ( pl# f -- )
3 IF
4 >R R R PIT C@ R DST C@
5 R> VOL C@ SOUND
6 ELSE
7 XSND
8 ENDIF ;
9 HEX
10 CODE DIG ( n -- n )
11 B5 C, 00 C, 94 C, 00 C,
12 94 C, 01 C, 38 C, A8 C,
13 36 C, 00 C, 36 C, 01 C,
14 88 C, D0 C, F9 C, 4C C,
15 NEXT , C; DCX -->

```

Screen: 66

```

0 ( Audio Editor )
1
2 : VOLUPD ( n -- )
3 4 0
4 DO
5 I STRIG
6 IF
7 DUP I VOL C@ + 0 MAX 15 MIN
8 I VOL C! I WVOL
9 ENDIF
10 LOOP
11 DROP ;
12
13
14
15 ==>

```

Screen: 69

```

0 ( Audio Editor )
1
2 : PDADJ ( hrz vrt pl# -- )
3 >R -DUP
4 IF 2* R DST C@ +
5 0 MAX 14 MIN R DST C!
6 R WDST
7 ENDIF
8 -DUP
9 IF I PIT C@ +
10 0 MAX 255 MIN R PIT C!
11 R WPIT
12 ENDIF
13 R> DROP ;
14
15 -->

```

Screen: 67

```

0 ( Audio Editor )
1
2 : AKEY ( -- n tf / ff )
3 0 764 C@ DUP 255 <
4 IF
5 255 764 C!
6 10 0
7 DO
8 DUP I TBL C@ =
9 IF
10 DROP NOT I SWAP 0 LEAVE
11 ENDIF
12 LOOP
13 ENDIF
14 DROP ;
15 -->

```

Screen: 70

```

0 ( Audio Editor )
1
2 : DIGMV ( pl# -- )
3 >R R PIT C@ 2/ 55 +
4 R DST C@ 4 * 21 +
5 R> PLYPUT ;
6
7
8
9
10
11
12
13
14
15 ==>

```

Screen: 68

```

0 ( Audio Editor )
1
2 : ?AKEY ( -- )
3 AKEY
4 IF
5 DUP 8 <
6 IF
7 ACTL C@ SWAP 1+ DIG XOR
8 ACTL C! WACTL
9 ELSE
10 9 = 2* 1- VOLUPD
11 ENDIF
12 ENDIF ;
13
14
15 ==>

```

Screen: 71

```

0 ( Audio Editor AUDED )
1
2 FORTH DEFINITIONS
3
4 : AUDED ( -- )
5 AUDPAL INIT
6 BEGIN 4 0
7 DO
8 I STICK I PDADJ
9 I DIGMV I I STRIG SND
10 LOOP
11 ?AKEY ?TERMINAL
12 UNTIL
13 OFF PLAYERS 0 752 C!
14 0 0 POS. XSND4 ;
15 BASE ! FORTH

```

Screen: 72

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 75

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 73

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 76

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 74

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 77

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 78

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 81

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 79

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 82

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 80

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 83

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 84

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 87

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 85

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 88

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 86

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 89

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 90

```

0 ( Charedit: var defs
1 BASE @ DCX
2 '( POS. )( : POS. 84 C! 85 ! ; )
3
4 '( STICK )( 84 KLOAD )
5
6 VOCABULARY CHREDT IMMEDIATE
7 CHREDT DEFINITIONS
8
9 0 VARIABLE HORZ
10 0 VARIABLE VERT
11 0 VARIABLE CHAR#
12 0 VARIABLE CURLOC
13 0 VARIABLE DEFLOC
14 0 VARIABLE TPTR
15 0 VARIABLE CSET-LOC ==>

```

Screen: 91

```

0 ( Charedit
1
2 : POSCUR ( n n -- )
3 SWAP CURLOC @
4 DUP C@ 84 -
5 SWAP C! 40 * + 203 +
6 88 @ + DUP C@
7 84 + OVER C!
8 CURLOC ! ;
9
10 : CLICK ( -- )
11 0 53279 C!
12 8 53279 C! ;
13
14
15 -->

```

Screen: 92

```

0 ( Charedit
1
2 HEX
3 : ANTIC ( f -- )
4 22F C@ SWAP
5 IF 20 OR ELSE DF AND ENDIF
6 22F C! ;
7
8 CODE CHSB0 ( b -- n )
9 B4 C, 00 C, C8 C, A9 C, 00 C,
10 95 C, 00 C, 95 C, 01 C, 38 C,
11 36 C, 00 C, 36 C, 01 C, 18 C,
12 88 C, D0 C, F8 C, 4C C, NEXT ,
13 C;
14 : CHSB1 ( n b -- f )
15 CHSB0 AND 0# ; DCX ==>

```

Screen: 93

```

0 ( Charedit
1 '( NFLG --> )( )
2
3 0 VARIABLE NFLG
4
5 : -NUMBER ( addr -- d )
6 BEGIN DUP C@ BL = DUP + NOT
7 UNTIL 0 NFLG ! 0 0 ROT DUP 1+
8 C@ 45 = DUP >R + -1
9 BEGIN DPL ! (NUMBER) DUP C@
10 DUP BL <> SWAP 0# AND
11 WHILE DUP C@ 46 - NFLG !
12 0 REPEAT DROP R> IF DMINUS
13 ENDIF NFLG @ IF 2DROP ENDIF
14 NFLG @ NOT NFLG ! ;
15 -->

```

Screen: 94

```

0 ( Charedit
1
2 : DSPCHR ( -- )
3 88 @ 203 + CURLOC ! DUP 320 +
4 SWAP
5 DO
6 I 8 0 DO
7 0 OVER C@ 7 I - CHSB1
8 IF 128 + ENDIF
9 CURLOC @ C! 1 CURLOC +!
10 LOOP
11 DROP 32 CURLOC +! 40
12 +LOOP 0 0 VERT ! HORZ ! 88 @
13 203 + DUP DUP CURLOC ! C@
14 84 + SWAP C! ;
15 ==>

```

Screen: 95

```

0 ( Charedit
1
2 : GRAFC ( -- n )
3 88 @ 882 + ;
4
5 : GR8 ( -- n )
6 88 @ 802 + ;
7
8 : SCR/W ( n n n -- )
9 SWAP B/SCR * OFFSET @ +
10 DUP 4 + SWAP
11 DO
12 2DUP I SWAP R/W
13 SWAP 128 + SWAP
14 LOOP
15 2DROP ; -->

```

Screen: 96

```

0 ( Charedit )
1 HEX
2 CODE CHSB2 ( n -- n )
3 B5 C, 00 C, 94 C, 00 C,
4 94 C, 01 C, 38 C, A8 C,
5 36 C, 00 C, 36 C, 01 C,
6 88 C, D0 C, F9 C, 4C C,
7 NEXT , C;
8 DCX
9
10 : MPTRR ( -- )
11 TPTR @ 0 OVER C! 1+ DUP
12 GR8 2- 33 + U>
13 IF 32 - ENDIF
14 DUP TPTR ! 93 SWAP C! CLICK ;
15 ==>

```

Screen: 97

```

0 ( Charedit )
1
2 : MPTRL ( -- )
3 TPTR @ 0 OVER C! 1-
4 DUP GR8 U<
5 IF
6 32 +
7 ENDIF
8 DUP TPTR !
9 93 SWAP C!
10 CLICK ;
11
12
13
14
15 -->

```

Screen: 98

```

0 ( Charedit )
1
2 HEX
3 : DBMAKE ( -- )
4 OFF ANTIC 58 @ 300 - DUP
5 58 ! FF00 AND DUP 230 !
6 DUP 3 70 FILL
7 3 + DUP 42 SWAP C!
8 1+ DUP 58 @ SWAP !
9 2+ DUP 15 2 FILL
10 15 + DUP 12 F FILL
11 12 + DUP 41 SWAP C!
12 1 + 230 @ SWAP !
13 ON ANTIC ;
14 DCX
15 ==>

```

Screen: 99

```

0 ( Charedit )
1
2 : PTCST ( scr# -- )
3 PAD CSET-LOC !
4 GRAFC DUP 320 + SWAP
5 2 0 DO
6 32 0 DO
7 DUP DUP 320 + SWAP DO
8 I C@ CSET-LOC @ C!
9 1 CSET-LOC +!
10 40 /LOOP
11 1+ LOOP
12 DROP
13 LOOP
14 PAD SWAP 0 SCR/W ;
15 -->

```

Screen: 100

```

0 ( Charedit )
1
2 : GTCST ( scr# -- )
3 GRAFC PAD ROT 1 SCR/W
4 PAD CSET-LOC ! 2 0
5 DO
6 32 0 DO
7 DUP DUP 320 + SWAP DO
8 CSET-LOC @ C@ I C!
9 1 CSET-LOC +!
10 40 /LOOP
11 1+ LOOP
12 288 + LOOP DROP GRAFC DUP
13 DEFLOC ! DSPCHR 0 CHAR# !
14 GR8 DUP 0 TPTR @ C! 12 14 POS.
15 0 . 93 SWAP C! TPTR ! ; ==>

```

Screen: 101

```

0 ( Charedit )
1
2 : GETSCR ( -- scr# )
3 BEGIN
4 18 14 POS. ." Screen #: "
5 PAD 5 EXPECT PAD 1- -NUMBER
6 DROP 128 17 C! 1 752 C!
7 18 14 POS. 16 SPACES NFLG @
8 IF
9 DUP 1 ( OVER 179 ) OR
10 ?1K IF OVER 89 ) OR ENDIF
11 IF DROP 0 ELSE 1 ENDIF
12 ELSE DROP 0
13 ENDIF
14 UNTIL
15 DUP 13 15 POS. 3 .R ; -->

```

Screen: 102

```

0 ( Charedit )
1
2 : VFIO ( -- f )
3 KEY 89 = 18 14 POS.
4 18 SPACES ;
5
6 : SVCST ( -- )
7 18 14 POS. ." Save this set?"
8 VFIO
9 IF GETSCR PTCST ENDIF ;
10
11 : LDCST ( -- )
12 18 14 POS. ." Load new set?"
13 VFIO
14 IF GETSCR GTCST ENDIF ;
15 ==>

```

Screen: 103

```

0 ( Charedit )
1
2 : MVRHT ( -- )
3 CHAR# @ DUP 63 <
4 IF
5 31 =
6 IF 289 ELSE 1 ENDIF
7 DEFLOC +!
8 1 CHAR# +! DEFLOC
9 @ DSPCHR MPTRR
10 12 14 POS.
11 CHAR# ?
12 ELSE
13 DROP
14 ENDIF ;
15 -->

```

Screen: 104

```

0 ( Charedit )
1
2 : MVLFT ( -- )
3 CHAR# @ -DUP
4 IF
5 32 =
6 IF -289 ELSE -1 ENDIF
7 DEFLOC +! -1 CHAR# +!
8 DEFLOC @ DSPCHR MPTRL
9 12 14 POS. CHAR# ?
10 ENDIF ;
11
12
13
14
15 ==>

```

Screen: 105

```

0 ( Charedit )
1
2 : CLRCHR ( -- )
3 DEFLOC @ 8 0
4 DO DUP I 40 * + 0 SWAP C! LOOP
5 DROP 88 @ 203 + 8 0
6 DO
7 DUP I 40 * + 8 0
8 DO
9 DUP I + 0 SWAP C!
10 LOOP DROP
11 LOOP DROP
12 0 VERT ! 0 HORZ !
13 88 @ 203 + DUP C@
14 84 + SWAP DUP
15 CURLOC ! C! ; -->

```

Screen: 106

```

0 ( Charedit )
1
2 : CLRCST ( -- )
3 18 14 POS. ." Clear this set?"
4 KEY 89 =
5 IF
6 GRAFC DUP DUP 680 + SWAP
7 DO
8 0 I C!
9 LOOP
10 CLRCHR 0 CHAR# ! DEFLOC !
11 12 14 POS. CHAR# ?
12 GR8 0 TPTR @ C! 93 OVER
13 C! TPTR !
14 ENDIF
15 18 14 POS. 15 SPACES ; ==>

```

Screen: 107

```

0 ( Charedit )
1
2 HEX
3
4 : CKOPT ( -- )
5 2FC C@ FF 2FC C!
6 DUP 1F = IF CLRCHR ENDIF
7 DUP 1E = IF CLRCST ENDIF
8 DUP 18 = IF LDCST ENDIF
9 DUP 1A = IF SVCST ENDIF
10 DUP 06 = IF MVLFT ENDIF
11 07 = IF MVRHT ENDIF ;
12
13
14
15 DCX -->

```


Screen: 108

```

0 ( Charedit )
1
2 : CKBTN ( -- )
3 644 C@ NOT
4 IF
5 CLICK
6 CURLOC @ DUP C@ 8 CHSB2 XOR
7 SWAP C! DEFLOC @ VERT @
8 40 * + DUP C@ 7 HORZ @
9 - 1+ CHSB2 XOR SWAP C!
10 2000 0 DO LOOP
11 ENDIF ;
12
13
14
15 ==>

```

Screen: 111

```

0 ( Charedit )
1
2 18 12 POS.
3 ." (4) Load a new set"
4 2 14 POS. ." Character 0"
5 2 15 POS. ." Load/Save: "
6 2 17 POS.
7 ." Use ' " 30 SPEMIT
8 ." ' and ' " 31 SPEMIT ." ' to"
9 CR
10 ." through the character set."
11 0 0 POS. ;
12
13
14
15 -->

```

Screen: 109

```

0 ( Charedit )
1
2 : CKSTK ( -- )
3 0 STICK 2DUP OR
4 IF
5 VERT @ + 0 MAX 7 MIN VERT !
6 HORZ @ + 0 MAX 7 MIN HORZ !
7 VERT @ HORZ @ POSCUR
8 2000 0 DO LOOP
9 ELSE
10 2DROP
11 ENDIF ;
12
13 : CHECK ( -- )
14 CKSTK CKBTN CKOPT ;
15 -->

```

Screen: 112

```

0 ( Charedit )
1
2 FORTH DEFINITIONS
3
4 : CHAR-EDIT ( -- )
5 CHREDT ( enter vocabulary )
6 0 GR. 1 752 C!
7 CLS DBMAKE
8 88 @ 1300 ERASE
9 GRAFC DEFLOC !
10 GR8 DUP TPTR !
11 93 SWAP C!
12 STPSCR
13 88 @ 203 + DUP CURLOC !
14 84 SWAP C!
15 ==>

```

Screen: 110

```

0 ( Charedit )
1
2 : STPSCR ( -- )
3 CR 4 SPACES
4 ." * * * CHARACTER-EDIT * * *"
5 CR CR CR ." 01234567" CR
6 8 0 DO I . CR LOOP
7 18 4 POS.
8 ." Options:"
9 18 6 POS.
10 ." (1) Clear Character"
11 18 8 POS.
12 ." (2) Clear this set"
13 18 10 POS.
14 ." (3) Save this set"
15 ==>

```

Screen: 113

```

0 ( Charedit )
1
2 0 HORZ !
3 0 VERT !
4 0 CHAR# !
5
6 DCX
7 BEGIN
8 CHECK
9 1 752 C! 128 17 C!
10 ?TERMINAL
11 UNTIL
12 0 GR. ;
13
14 BASE ! FORTH
15

```

Screen: 114

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 117

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 115

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 118

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 116

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 119

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 120

```
0 ( Character words:  CHLOAD      )
1
2 BASE @ DCX
3
4 : CHLOAD      ( addr scr# cnt -- )
5   8 * DUP <ROT
6   128 /MOD SWAP 0# +
7   >R B/SCR * R) 0
8   DO
9     PAD 128 I * +
10    OVER I + 1 R/W
11    LOOP
12    DROP
13    PAD <ROT CMOVE ;
14
15                                ==>
```

Screen: 123

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 121

```
0 ( Character words:  NML/SPLCHR )
1
2
3 : SPLCHR      ( CHBAS -- )
4   SP@ 1+ C@
5   SWAP DROP 756 C! ;
6
7
8 : NMLCHR      ( -- )
9   57344 SPLCHR ;
10
11
12 BASE !
13
14
15
```

Screen: 124

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 122

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 125

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 126

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 129

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 127

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 130

0
1
2
3
4
5
6
7 (Standard Character set)
8
9
10
11
12
13
14
15

Screen: 128

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 131

0
1
2
3
4
5
6
7 (Standard Character set)
8
9
10
11
12
13
14
15

Screen: 132

0
1
2
3
4
5
6
7 (PM example #2 ship images)
8
9
10
11
12
13
14
15

Screen: 135

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 133

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 136

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 134

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 137

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 138

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 141

0 (Player/Missile example 1)
1
2 : BOP 0 53279 C! 8 53279 C! ;
3
4 : MOVE-BALL
5 BEGIN
6 HBALL @ VBALL @ 0 PLYMV
7 0 PLYSTT C@ DUP 3 AND
8 IF VBALL @ MINUS VBALL ! BOP
9 ENDIF
10 3)
11 IF HBALL @ MINUS HBALL ! BOP
12 ENDIF
13 50 0 DO LOOP (Wait...)
14 ?TERMINAL
15 UNTIL ; -->

Screen: 139

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 142

0 (Player/Missile example 1)
1
2 : BOUNCE
3 CLS
4 1 PMINIT
5 PMCLR
6 1 PRIOR
7 ON PLAYERS
8 47 200 32 217 0 PLYBND
9 0 9 (BLUE) 8 PMCOL
10 IMAGE 7 100 75 0 BLDPLY
11
12 ." Press START to stop... "
13 MOVE-BALL
14 OFF PLAYERS ;
15 BASE !

Screen: 140

0 (Player/Missile example 1)
1 '(PLYMV)(15 KLOAD)
2 BASE @ 2 BASE !
3
4 1 VARIABLE HBALL
5 1 VARIABLE VBALL
6
7 LABEL IMAGE
8 011100 C,
9 111110 C,
10 111110 C,
11 111110 C, (A BIG BALL)
12 111110 C,
13 111110 C,
14 011100 C,
15 DECIMAL ==>

Screen: 143

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 144

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 147

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 145

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 148

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 146

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 149

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 150

```

0 ( Player/Missile example 2 )
1 BASE @ DCX
2 '( CHLOAD )( 60 KLOAD )
3 '( PLYMV )( 15 KLOAD )
4 '( STICK )( 84 KLOAD )
5 : FLY
6 BEGIN
7 75 @ DO LOOP ( wait )
8 PAD ( addr )
9 @ PLYLOC SWAP DROP
10 8 / 11 SWAP -
11 11 MIN @ MAX ( image# )
12 @ PLYSEL ( pl#0 )
13 @ STICK @ PLYMV
14 ?TERMINAL
15 UNTIL ; ==>

```

Screen: 153

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 151

```

0 ( Player/Missile example 2 )
1
2 : SHIP
3 2 PMINIT
4 1 PRIOR
5 PMCLR
6 @ 9 ( BLUE ) 8 PMCOL
7 PAD 132 15 CHLOAD
8 PAD 8 50 50 @ BLDPLY
9 50 200 10 110 @ PLYBND
10 CLS
11 ." Move player with stick 0."
12 CR
13 ." Press START to stop... "
14 ON PLAYERS FLY OFF PLAYERS ;
15 BASE ! -->

```

Screen: 154

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 152

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 155

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```


Screen: 156

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 159

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 157

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 160

```
0 ( Utils: CARRAY ARRAY )
1 BASE @ HEX
2 : CARRAY ( cccc, n -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ALLOT
5 ;CODE CA C, CA C, 18 C,
6 A5 C, W C, 69 C, 02 C, 95 C,
7 00 C, 98 C, 65 C, W 1+ C,
8 95 C, 01 C, 4C C,
9 ' + ( CFA @ ) , C;
10
11 : ARRAY ( cccc, n -- )
12 CREATE SMUDGE ( cccc: n -- a )
13 2* ALLOT
14 ;CODE 16 C, 00 C, 36 C, 01 C,
15 4C C, ' CARRAY 08 + , C; ==>
```

Screen: 158

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 161

```
0 ( Utils: CTABLE TABLE )
1
2 : CTABLE ( cccc, -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ;CODE
5 4C C, ' CARRAY 08 + , C;
6
7 : TABLE ( cccc, -- )
8 CREATE SMUDGE ( cccc: n -- a )
9 ;CODE
10 4C C, ' ARRAY 0A + , C;
11
12
13
14
15
```

Screen: 162

```

0 ( Utils: 2CARRAY 2ARRAY )
1
2 : 2CARRAY ( cccc, n n -- )
3 (BUILDS ( cccc: n n -- a )
4 SWAP DUP , * ALLOT
5 DOES)
6 DUP >R @ * + R) + 2+ ;
7
8 : 2ARRAY ( cccc, n n -- )
9 (BUILDS ( cccc: n n -- a )
10 SWAP DUP , * 2* ALLOT
11 DOES)
12 DUP >R @ * + 2* R) + 2+ ;
13
14
15 ==>

```

Screen: 163

```

0 ( Utils: XC! X! )
1
2 : XC! ( n0...nm cnt addr -- )
3 OVER 1- + >R 0
4 DO J I - C!
5 LOOP R) DROP ;
6
7 : X! ( n0...nm cnt addr -- )
8 OVER 1- 2* + >R 0
9 DO J I 2* - !
10 LOOP R) DROP ;
11
12 ( Caution: Remember limitation
13 ( on stack size of 30 values
14 ( because of OS conflict. )
15 -->

```

Screen: 164

```

0 ( Utils: CVECTOR VECTOR )
1
2 : CVECTOR ( cccc, cnt -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 HERE OVER ALLOT XC!
5 ;CODE
6 4C C, ' CARRAY 0B + , C;
7
8 : VECTOR ( cccc, cnt -- )
9 CREATE SMUDGE ( cccc: n -- a )
10 HERE OVER 2* ALLOT X!
11 ;CODE
12 4C C, ' ARRAY 0A + , C;
13
14
15 BASE !

```

Screen: 165

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 166

```

0 ( Sound: SOUND SO. FILTER! )
1
2 BASE @ HEX
3 0 VARIABLE AUDCTL
4
5 : SOUND ( ch# freq dist vol -- )
6 3 DUP D20F C! 232 C!
7 SWAP 10 * + ROT 2*
8 D200 + ROT OVER C! 1+ C!
9 AUDCTL C@ D208 C! ;
10
11 : SO. SOUND ;
12
13 : FILTER! ( b -- )
14 DUP D208 C! AUDCTL ! ;
15 ==>

```

Screen: 167

```

0 ( Sound: XSND XSND4 )
1
2
3 : XSND ( voice# -- )
4 2* D201 +
5 0 SWAP C! ;
6
7
8 : XSND4 ( -- )
9 D200 8 0 FILL
10 0 FILTER! ;
11
12
13 ' ( POS. ) ( : POS. 54 C! 55 ! ; )
14
15 BASE !

```

Screen: 168

```
0 ( Utils: STICK )
1 BASE @ HEX
2 LABEL STKARY
3 0 , -1 , 1 , 0 ,
4
5 : STICK ( n -- n n )
6 278 + C@ @F XOR
7 DUP 2/ 2/ 3 AND
8 2* STKARY + @
9 SWAP 3 AND
10 2* STKARY + @ ;
11
12 CODE STRIG ( n -- f )
13 B4 C, 00 C, B9 C, 284 ,
14 49 C, 01 C, 4C C, PUT0A , C;
15 BASE !
```

Screen: 171

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 169

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 172

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 170

```
0 CONTENTS OF THIS DISK:
1
2 PLAYER/MISSILES: 30 LOAD
3 AUDIO EDITOR: 60 LOAD
4 CHARACTER EDITOR: 90 LOAD
5 CHARACTER SET WORDS: 120 LOAD
6
7 STANDARD CHARACTER SET 130 LIST
8 SPACE SHIP IMAGES 132 LIST
9
10 PM EX. #1 ( BOUNCE ) 140 LOAD
11 PM EX. #2 ( SHIP ) 150 LOAD
12
13 ARRAYS ( FOR ALL ) 160 LOAD
14 SOUNDS ( FOR AUDED ) 166 LOAD
15 STICK 168 LOAD
```

Screen: 173

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages)
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 179

0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

DISPLAY FORMATTER

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

Software and Documentation
© Copyright 1982
Valpar International

vaIFORTH
T.M.

DISPLAY FORMATTER

Version 1.1
March 1982

The following is a description of commands used in creating video display lists on the Atari 400/800 series microcomputers. Creating custom display lists allows for innovative graphic layouts of games, simulations, or business applications which utilize both hi-resolution graphics and text simultaneously.

valFORTH^{T.M.}
SOFTWARE SYSTEM

DISPLAY FORMATTER

Stephen Maguire

Software and Documentation

© Copyright 1982

Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

An indepth explanation of display lists was written by Dave and Sandy Small in a series of articles found in Creative Computing. We suggest that this be read to get the most out of this valFORTH package.

STROLLING THROUGH THE DISPLAY FORMATTER

In Atari Basic there are many different graphic modes. Some of these are text modes, some are graphics modes, and some are mixed. These different graphic modes are based upon display lists. A display list is a list of display instructions which tell the video processor whether a particular portion of the screen is to be high resolution graphics or normal text. Any given section of the display can actually take on one of 18 different characteristics.

Let's take a look at the display list for a graphic 0 display: (These values are in base 16)

BC20	70}	
	70}	24 blank scan lines
	70}	
	42}	
	40}	DM jump to BC40
	BC}	
	2}	
	2}	
	.	
	.	23 graphic 0 lines
	.	
	2}	
	2}	
	41}	
	20}	jump to BC20
	BC}	
BC40		start of display memory

Each opcode 70 instructs the video processor to display 8 blank scan lines. Opcode 2 produces one standard graphic 0 text line. Opcode 42 is a modified 2 instruction. In addition to creating a standard text line, it also informs the video processor where the display memory is located (the address is found in the next two bytes). At the end of the list there is a three byte jump instruction which transfers display list interpretation to the address specified in the next two bytes of the list. Each of the graphic settings have a similar list. This valFORTH package allows you to design your own lists. Let's make one now.

Look in the directory (screen 170) and load in the display formatter. Most of the formatter words begin with DB (for display block). To initialize the system type:

```
DBINIT
HEX
```

valFORTH Display Formatter 1.1

This initializes the system and puts it into the more useful hexadecimal mode. Graphic mode 8 is a high resolution graphic mode with a four line text window at the bottom of the display. Let's make a display with a four line text window at the top of the screen followed by the high resolution graphics plate. First, we need 24 blank scan lines at the top:

```
70 DBM
70 DBM
70 DBM
```

The DBM command stands for "Display-Block Make." It takes the opcode on top of the stack and tacks it onto the end of the display list currently being created. Additionally, it enters an address into the array DBLST which points to the first byte of memory used by that display block. There is a plural form of the DBM command:

```
3 70 DBMS
```

This adds 3 opcode 70's to the current display list. Now let's add the four line text window. Recall that a normal text line has an opcode of two:

```
4 2 DBMS
```

Note that the display memory jump described earlier is automatically inserted into the display list. Now we need to define the high resolution portion of the display. A standard graphic 8 line has an opcode of \$F (15 in decimal). Let's create 20 graphic 8 lines (20 in base 16 is 14).

```
14 F DBMS
```

This list is good enough for now. To verify that it has been entered properly, type:

```
DMPLST
```

You should get something like:

BLK	ADDR	BYTE	MODS
---	----	----	----
0	A100	70	
1	A100	70	
2	A100	70	
3	A100	2	J A100
4	A128	2	
5	A150	2	
6	A178	2	
7	A1A0	F	
8	A1C8	F	
9	A1F0	F	
A	A218	F	
B	A240	F	
C	A268	F	
D	A290	F	
E	A2B8	F	
F	A2E0	F	
10	A308	F	
11	A330	F	
12	A358	F	
13	A380	F	
14	A3A8	F	
15	A3D0	F	
16	A3F8	F	
17	A420	F	
18	A448	F	
19	A470	F	
1A	A498	F	

Note the automatic insertion of the display memory jump in block three. Display memory cannot cross a 4K memory boundary without a display memory jump. As each display block is added, a check is made to detect any 4K memory crossings caused by the display block. If the block does cross, a display memory jump is automatically inserted into the list to account for it.

Now that we have a display list, let's enable it. There are several ways to activate a list. For now type:

MIXED CLS

This MIXED command enables the new display list and also re-directs output to the display memory specified by the list. This allows for interactive display list creation. There should be a recognizably different display. Hold down the RETURN key and watch how the "ok" message is displayed as the cursor

moves down the screen. You should see "ok"'s on the text lines, but in the high resolution lines, it should look quite different. You can type in a high resolution mode because the Atari operating system does not know that the display list has been changed. To return to a normal display, the GR. command is used:

0 GR.

Dump the display list again using the DMPLST command. Let's put some text lines in at block B. To do this type:

B DBPTR
10 2 DBMS

The DBPTR command positions the display list pointer to the specified block. That block then becomes the end of the list. After that, we add 16 (10 hex) graphic 0 lines. Dump the list again and verify that this is indeed what was accomplished. To view this new display, type:

MIXED CLS

Hold down the RETURN key again. Notice what happens as the cursor passes through the high resolution section and then back into the second text section. Type DMPLST again while in this mode and notice that everything works the same, the data is simply displayed differently. To get out, type "0 GR."

Besides adding display blocks onto the end of a display list, the display formatter allows display blocks to be inserted and deleted as well. Block two has an opcode 70 which produces 8 blank scan lines on the video screen. By deleting this block from the list, the entire display will shift upwards by 8 lines. This is accomplished using the DBDEL command:

2 DBDEL

Dump the list and verify that the block has indeed been deleted. Enable the list using "MIXED CLS". Note that the first text line appears much higher than usual on the video screen. While still in this display, execute:

4 6 DBDELS

This will delete the four display blocks starting at block six. In this case, the four high resolution display lines are deleted. Type "MIXED CLS" and watch the screen shrink slightly as the display blocks are extracted.

Display blocks can be inserted using the DBIN command. When a DBIN command is executed, the specified opcode is inserted into the specified block. The opcode previously in that block and all opcodes following are pushed back by one block. As an example, we will insert opcode 70 (8 blank scan lines) at block five. This will do it:

70 5 DBIN

"MIXED CLS" will activate the new list. Press the RETURN key a few times and notice how the output routines seem to ignore the blank scan lines. The DBINS command is a plural form of the DBIN command. Let's insert a different opcode other than 2 or \$F. Opcode 6 is a mode which displays colored characters which are much larger than normal. This will insert three opcode 6's at block 9:

3 6 9 DBINS

Activate this new list in the normal way and experiment with it. The following section describes all of the available opcodes. Experiment with these as you read about them and you should have no problem understanding any of them.

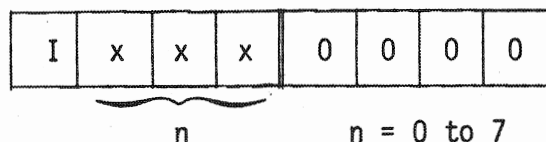
This brief explanation of display list formatting should show the power available to the programmer who wants to get that unique display. There are many more commands available for use. These are explained thoroughly in the glossary at the end of the next section.

DISPLAY LIST INSTRUCTIONS

There are four basic display list instructions. Those that produce blank scan lines, the display list jump, the jump on vertical blank, and the display block instructions. This is a description of these four basic instructions.

Blank Scan Lines

Byte form:



This opcode produces n+1 blank scan lines of color BAK. No video memory is used by this instruction.

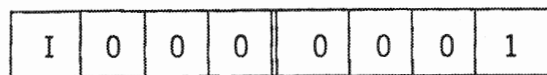
If the I bit is set, a display list interrupt (DLI) will occur upon interpretation by Antic (the video processor).

The 8 legal values are:

\$00 = 0	1 blank scan line (128 with I bit set)
\$10 = 16	2 blank scan lines (144)
\$20 = 32	3 blank scan lines (160)
\$30 = 48	4 blank scan lines (176)
\$40 = 64	5 blank scan lines (192)
\$50 = 80	6 blank scan lines (208)
\$60 = 96	7 blank scan lines (224)
\$70 = 112	8 blank scan lines (240)

Display List Jump

Byte form:



This command instructs Antic to search for the next display list instruction specified by the address contained in the next two bytes of the display list. The low byte of the address is found lower in memory. This command is used primarily to continue a display list across a 1K memory boundary (Antic will not handle this properly). This is the only instruction not supported by the display formatter since its occurrence is rare. It is explained here for completeness sake and its use is absolutely forbidden. Future releases may have this implemented.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

Legal form:

\$01 addr-low addr-hi Transfer display list
interpretation to addr.

Jump On Vertical Blank

Byte form:

I	1	x	x	0	0	0	1
---	---	---	---	---	---	---	---

This three byte opcode instructs Antic to transfer display list interpretation to the address specified by the following two bytes (low byte of address first) and to pause until vertical blank occurs. Since display list processing halts, any remaining portion of the video display takes on the color of BAK. This command is not to be entered by the user. The display formatter automatically adds this to the end of the display list whenever it is moved or activated.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

Legal form:

\$41 addr-low addr-hi Transfer display list
interpretation to addr. (65)

Display Block Opcodes

Byte form:

I	J	V	H	x	x	x	x
---	---	---	---	---	---	---	---

n = 2 to \$F (15)

n

There are 14 display modes. Six are character modes, eight are graphic modes. Each of these modes varies greatly and will be discussed individually. But first, the four status bits I, J, V, and H, will be discussed as they function similarly for all display modes.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

If set, the J bit instructs Antic to perform a display memory jump. Antic expects the next two bytes in the display list to point to the new display memory location. The first display block instruction should always have this bit set. Also, Antic cannot properly retrieve data from display memory across 4K boundaries. Thus, if the display memory must cross a 4K boundary, a display memory jump must be used. Note that the display formatter automatically takes care of these two problems for the user.

If set, the V bit informs Antic that the current display block is to be vertically scrolled upward according to the value in VSCROL (address \$D405). Note that vertical scrolling is accomplished only if two or more consecutive display blocks have this bit set.

If set, the H bit informs Antic that the current display block is to be horizontally scrolled right according to the value in HSCROL (address \$D404). Note that for horizontally scrolled display blocks, extra bytes of memory are needed. The exact number of bytes varies for different screen (playfield) widths. Use the following calculation:

$$\# \text{ extra} = X / n$$

where: X = the number of characters/display block
 n = 4 for a narrow playfield
 = 5 for a standard playfield

There are no extra bytes for the wide playfield setting.

For example, a 40 character/line display block in the standard width would use a total of $40 + 40/5$ or 48 characters. Note that only one of these extra bytes is actually used for the display.

The Character Modes

There are 6 character modes (opcodes 2 thru 7). All character modes work in the same way, i.e., the values in display memory are indices to a large "n" by 8 byte array. In some of these modes, the highest one or two bits are used to specify a color with only the remaining lower bits used for indexing. The following table gives information about each of the modes:

Antic mode	2	3	4	5	6	7
Basic mode	0	---	---	---	1	2
# color *	1.5	1.5	5	5	5	5
Chars/line narrow wid	32	32	32	32	16	16
Chars/line normal wid	40	40	40	40	20	20
Chars/line wide screen	48	48	48	48	24	24
Scan lines/ pixel	8	10	8	16	8	16
Bits/pixel	1	1	2	2	1	1
Color clocks per pixel	.5	.5	1	1	1	1

Colors:

- mode 2: Takes the color of PF2 with the lum of PF1
(Artifacting/bleed very noticeable)
- mode 3: Same as above
- mode 4: Two bits/pixel in character definitions
00 = BAK 01 = PF0 10 = PF1
11 = PF2 if bit 7 of index = 0, else PF3
- mode 5: Same as 4 above
- mode 6: Most significant two bits of index
0 = PF0 1 = PF1 etc.
- mode 7: Same as 6 above

The Graphic Modes

There are 8 graphic modes. Unlike character modes, the values in display memory are not indices into an array of character definitions, but rather are the definitions themselves. Depending on the graphic mode, these values give different results. The following table gives various information about each mode.

Antic mode	8	9	A	B	C	D	E	F*
Basic mode	3	4	5	6	---	7	---	8
# colors	4	2	4	2	2	4	4	1.5
bytes/line narrow wid	8	8	16	16	16	32	32	32
bytes/line normal wid	10	10	20	20	20	40	40	40
bytes/line wide screen	12	12	24	24	24	48	48	48
Pixels per normal wid	40	80	80	160	160	160	160	320
Scan lines/pixel	8	4	4	2	1	2	1	1
Bits/pixel	2	1	2	1	1	2	2	1
Color clocks per pixel	4	2	2	1	1	1	1	.5

*Mode F values differ when in GTIA modes

Colors:

mode 8: Two bits/pixel, 4 pixels/byte
00 = BAK 01 = PFO 10 = PF1 11 = PF2
mode 9: One bit/pixel, 8 pixels/byte
0 = BAK 1 = PFO
mode A: Same as mode 8 above
mode B: Same as mode 9 above
mode C: Same as mode 9 above
mode D: Same as mode 8 above
mode E: Same as mode 8 above
mode F: Take the color of PF2 and lum of PF1
(if not in a GTIA mode)

GLOSSARY

(DBINIT)

(dmem dlist ---)

The (DBINIT) routine initializes the display formatter. It expects two addresses on the stack. The address on top of the stack is used as the target address for the display list. The address found second on the stack is the target address for display memory. The display list is actually created in a c-array named DSPLST. Note that while building the list, no check is made to ensure that the display list does not cross a 1K memory boundary.

DBINIT

(---)

Like the (DBINIT) command above, this initializes the display formatter. But unlike (DBINIT), this expects no arguments. Instead, these values are calculated automatically. The display memory address is top of memory minus 1F00 hex. This is enough for a full graphics 8 screen. The display list address is 256 bytes below the display memory address. Note that this is very memory wasteful, and should only be used while still learning the system. After that, (DBINIT) should be used.

DBPTR

(block# ---)

This command instructs the display formatter to create the next display block in the specified "block#" of the current display list. To begin creating a new display list, use:

0 DBPTR

DBM

(antic-mode ---)

The DBM command adds "antic-mode" to the end of the current display list. For example, to create a video display with a single line at the top of the screen, the following would be executed:

```
0 DBPTR    (new list)
2 DBM      (A graphic 0 line)
```

(Note: Antic mode 2 is a BASIC graphics 0 line.)

DBMS

(#times antic-mode ---)

The DBMS command performs a multiple DBM. For example, to create a full graphics 0 screen, the following two commands must be performed:

```
0 DBPTR    (new list)
24 2 DBMS  (24 graphic 0 lines)
```

This would create a full graphics 8 screen:

```
0 DBPTR
192 15 DBMS    (Antic 15 = graphic 8)
```

(192 graphic 8 lines fill one video screen)

DBMS (cont'd)

Mixed lists are also possible:

```

      0 DBPTR
160 15 DBMS
      4  2 DBMS
    
```

This would create a screen of 160 graphic 8 lines with four text lines at the bottom.

DBIN

(antic-mode block# ---)

Oftentimes, it is desirable to slightly change the existing display list to obtain special effects midway through a running program. The DBIN command allows insertion of new display blocks within the current display list. This command inserts "antic-mode" into the block specified by "block#". Whatever was in the block "block#" and following is pushed back one block. For example:

Display list

block #0	2
1	2
2	8

with the above display list, a

```
15 1 DBIN
```

would give the following display list.

Display list

block # 0	2
1	15
2	2
3	8

The DBIN allows the user to create new display lists without the need to duplicate already existing display list sections.

DBINS (#times antic-mode block# ---)

This command repeats "antic-mode block# DBIN" the specified number of times.

DBDEL (block# ---)

The DBDEL command serves as the logical complement to the DBIN command. Thus, after inserting a temporary display block, the DBDEL command may be used to delete that display block once it is no longer needed:

Display list	
block # 0	2
1	15
2	2
3	8

1 DBDEL would give:

Display list	
block # 0	2
1	2
2	8

Note: Deleting non-existing display blocks gives unexpected results.

DBDELS (#times block# ---)

This command performs "block# DBDEL" the specified number of times. This serves as the logical complement to the DBINS command.

DBDELL

(---)

This form of the DBDEL command deletes the last display block created using the DBM command. For example:

Display list	
block # 0	2
1	15
2	2
3	8

DBDELL would give:

Display list	
block # 0	2
1	15
2	2

The main use for the DBDELL command is for "backing up" and re-entering a display block when an error has been made while creating a display list directly at the keyboard. The DBDELL command can be used successively for deleting a section of display blocks at the end of the current display list. There is no plural command for DBDELL command as its use is rather limited.

?ANTMOD

(block# --- antic-mode)

Occasionally, it is desirable to know what antic-mode is being used for a particular display block (such as for a text output routine -- text should not be output on a hi-resolution line, for example). This command returns the antic-mode of the specified block.

DBMOD

(modifier block# ---)

When creating display lists, it is possible to give extra meaning to a particular block or section of blocks in the list. This is accomplished by using one or more of the three available antic-modifiers: vertical scroll modifier (VRTMOD), horizontal scroll modifier (HRZMOD), and the display-list interrupt (INTMOD). The following are examples of each:

```
VRTMOD 0 DBMOD
HRZMOD 3 DBMOD
INTMOD 5 DBMOD
```

DBMOD (cont'd)

There are several methods in which to put more than one modifier on a given display block. For example, each of the following would give the same final result:

```
VRTMOD 20 DBMOD
HRZMOD 20 DBMOD
```

or

```
VRTMOD HRZMOD + 20 DBMOD
```

To attach all three modifiers, the best method is:

```
VRTMOD HRZMOD INTMOD + + 20 DBMOD
```

It should also be noted that it is possible to create modified display blocks, thus reducing the need for the DBMOD command:

```
HRZMOD 2 + DBM
```

This would create one graphic 0 line with a horizontal modifier. It is also easy to obtain 16 lines of hi-resolution graphics with both horizontal and vertical scroll modifiers:

```
16 VRTMOD HRZMOD 15 + + DBMS
```

CAUTION: VRTMOD and HRZMOD can only be used on antic-modes 2 through 15 (\$2-\$F).

(Note: There is one additional modifier, JMPMOD; however its use is absolutely forbidden! This has been defined as it will be implemented in the next release.)

DBMODL (modifier ---)

This command modifies the last display-block in the display list.

?DBMODS (block# --- modifiers)

This returns the modifiers on the specified display block. For example:

```
VRTMOD 2 + 0 DBM
0 ?DBMODS
```

would give VRTMOD. Also:

```
VRTMOD HRZMOD 2 + + 0 DBM
0 ?DBMODS
```

would give VRTMOD + HRZMOD. To test for VRTMOD, the following method must be used:

```
0 ?DBMODS
VRTMOD AND
```

The last line leaves only the vertical modifier, if present, or leaves 0 indicating no vertical modifier.

DBREM (block# ---)

The DBREM command removes all modifiers from the specified display block. Care should be taken when stripping modifiers, as stripping a horizontal modifier (if present) will change the size of the video memory.

DBREMS (#times block# ---)

This performs "block# DBREM" the specified number of times.

DBREML (---)

This removes the modifiers from the last display block in the current display list.

?DBVAL (block# --- info)

The ?DBVAL command returns all information about the display block specified, i.e., the antic mode and any modifiers. This information is returned as one value.

DBWID (width ---)

The DBWID command is used to set the desired playfield width so that the address array DBLST gives the proper values. Legal settings are: 1 - narrow, 2 - normal, and 3 - wide.

USRDSP (---)

Once a display list has been created, USRDSP activates the new list.

MIXED (---)

The MIXED command performs a USRDSP then instructs the Atari operating system to re-direct all output to the video display memory specified by the newly created display list.

DMPLST (---)

The DMPLST command instructs the display list assembler to give a complete, informative listing of the display list last created.

DBADR (block# --- address)

The DBADR command is one of the most useful commands to the programmer. Given a display block number, it returns the address of the first byte of that display block. This is extremely useful for determining where output text or graphic displays should be located.

DMCLR (---)

The DMCLR command clears the display memory pointed to by the display list currently being created. It clears to the top of memory.

In addition, there are various variables available to the programmer:

DSPEND	Points to the end of the current display list. It is an offset from 0 DSPLST.
DSPBLK	Contains the number of the next display block to be created.
DMLOC	Points to the beginning of display memory.
LSTLOC	Contains the address of where the display list is to reside in memory.
DBLST	Is an array of addresses used by DBADR.
DSPLST	Is a byte array containing the display list currently being created. DSPEND above points to the end of the list in this array.

XXXII. DISPLAY FORMATTER SUPPLIED SOURCE LISTING

Screen: 30

```

0 ( Graph Sys: tables )
1 '( TRANSIENT TRANSIENT )( )
2 BASE @ DECIMAL
3 '( CTABLE )( 45 KLOAD )
4
5 LABEL BLKNML
6 0 C, 0 C, 40 C, 40 C, 40 C,
7 40 C, 20 C, 20 C, 10 C, 10 C,
8 20 C, 20 C, 20 C, 40 C, 40 C,
9 40 C,
10
11 CTABLE HSOFS
12 50 C, 4 C, 5 C, 50 C,
13
14 TABLE BLKOFS
15 50 , -5 , 50 , 5 ,      ==>

```

Screen: 33

```

0 ( Graph Sys: [DBINIT] DBINIT )
1
2
3 : (DBINIT)      ( DM LIST -- )
4 LSTLOC !      DUP
5 DMLOC !      0 DBLST !
6 0 DSPEND !    0 PGECS !
7 0 DSPBLK !    1 NWLST !
8 0 JMPDAT 11 ERASE ;
9
10 : DBINIT      ( -- )
11 106 C@ 256 * 7936 -
12 DUP 256 -
13 (DBINIT) ;
14
15 DBINIT      -->

```

Screen: 31

```

0 ( Graph Sys: variables )
1
2 CTABLE BYTBK 16 ALLOT
3 BLKNML 0 BYTBK 16 CMOVE
4
5 255 CARRAY DSPLST
6 255 ARRAY DBLST
7 5 VARIABLE HS# /
8 0 VARIABLE DSPBLK
9 0 VARIABLE PGECS
10 0 VARIABLE DSPEND
11 0 VARIABLE NWLST
12 6 VARIABLE SDTMP
13 0 VARIABLE DBCNT
14 0 VARIABLE DBVRT
15      -->

```

Screen: 34

```

0 ( Graph Sys: HINYB EODB )
1 DECIMAL
2
3 : HINYB      ( nmmm -- n )
4 61440 AND 4096 / 16 + 15 AND ;
5
6 : EODB      ( n -- a )
7 31 AND DUP 15 AND ( Find end )
8 BYTBK C@ SWAP ( of disp )
9 15 )      ( block )
10 IF      ( Horz. scroll )
11 DUP HS# / @ / +
12 ENDIF
13 DSPBLK @      ( Update the )
14 DBLST @      ( addr list )
15 + 1- ;      ==>

```

Screen: 32

```

0 ( Graph Sys: constants )
1 HEX
2 0 VARIABLE DMLOC
3 0 VARIABLE LSTLOC
4
5 10 CONSTANT HRZMOD
6 20 CONSTANT VRTMOD
7 40 CONSTANT JMPMOD
8 80 CONSTANT INTMOD
9
10 DECIMAL
11
12 11 CARRAY JMPDAT
13 10 CARRAY JMPSTT
14
15 0 JMPDAT 11 ERASE      ==>

```

Screen: 35

```

0 ( Graph Sys: DBPTR )
1
2 : DBPTR      ( blk# -- )
3 DMLOC @ 0 DBLST !
4 0 PGECS !    0 NWLST !
5 DUP DSPBLK !    DUP DSPEND !
6 0 JMPDAT C@ -DUP
7 IF 1+ 1 DO
8 I JMPDAT C@ OVER (
9 IF
10 2 DSPEND +!
11 ELSE
12 I 1- 0 JMPDAT C! LEAVE
13 ENDIF
14 LOOP
15 ENDIF DROP      -->

```

Screen: 36

```

0 ( Graph Sys: JMPINS DBPTR )
1
2 DSPEND @ DSPBLK @ =
3 IF
4 1 NWLST !
5 ELSE
6 DSPBLK @ DBLST @
7 1- HINYB PGE CRS !
8 ENDIF ;
9
10 : JMPINS ( n -- )
11 DUP JMPMOD OR
12 DSPEND @ SWAP OVER
13 DSPLST C! 1+ NWLST @
14 IF
15 0 NWLST ! ==>

```

Screen: 37

```

0 ( Graph Sys: JMPINS )
1
2 DMLOC @ DUP DUP 40 +
3 HINYB SWAP HINYB < >
4 IF
5 HINYB 1+ 4096 *
6 ENDIF
7 ELSE
8 PGE CRS @ 4096 *
9 ENDIF
10 DUP ROT DSPLST !
11 3 DSPEND +!
12 DSPBLK @ 0 JMPDAT C@ 1+
13 DUP 0 JMPDAT C!
14 JMPDAT C! ;
15 -->

```

Screen: 38

```

0 ( Graph Sys: DBCRT )
1
2 : DBCRT ( n -- )
3 DUP EODB DUP
4 HINYB PGE CRS @
5 OVER PGE CRS ! < >
6 IF
7 DROP JMPINS
8 DSPBLK @ DBLST ! EODB
9 ELSE
10 SWAP DSPEND @
11 DSPLST C!
12 1 DSPEND +!
13 ENDIF
14 1+ DSPBLK 1 OVER +!
15 @ DBLST ! ; ==>

```

Screen: 39

```

0 ( Graph Sys: DBM )
1
2 : DBM ( n -- )
3 DUP 15 AND
4 IF ( antic instr.? )
5 191 AND ( strip jump )
6 DBCRT
7 ELSE ( scan lines )
8 DSPEND @
9 DSPLST C!
10 1 DSPEND +!
11 DSPBLK @ DUP DBLST @
12 SWAP 1+ DBLST !
13 1 DSPBLK +!
14 ENDIF ;
15 -->

```

Screen: 40

```

0 ( Graph Sys: LSTSV )
1
2 : LSTSV ( blk# -- a )
3
4 DSPEND @ ( pt to blk )
5 DSPLST 65 OVER C!
6 SWAP DBPTR
7
8 DSPEND @ DSPLST ( save list )
9 DUP SDTMP @ + ROT
10 >R OVER R> -
11 ABS 1+ <CMOVE
12
13 DSPEND @ DSPLST ( leave save )
14 SDTMP @ + ; ( address )
15 ==>

```

Screen: 41

```

0 ( Graph Sys: LSTRST )
1
2 : LSTRST ( a -- )
3 BEGIN
4 DUP C@ DUP 65 < > ( eolst? )
5 WHILE
6 DUP 15 AND 0#
7 OVER 64 AND 0# AND
8 IF ( jump? )
9 191 AND DBM 3 +
10 ELSE ( normal )
11 DBM 1+
12 ENDIF
13 REPEAT
14 2DROP ;
15 -->

```

Screen: 42

```

0 ( Graph Sys: DBIN DBDEL DBMOD )
1
2 : DBIN ( n n -- )
3 LSTSV SWAP DBM LSTRST ;
4
5 : DBDEL ( n -- )
6 DUP 1+ LSTSV SWAP
7 DBPTR LSTRST ;
8
9 : DBMOD ( n n -- )
10 DUP 1+ LSTSV
11 SWAP DBPTR SWAP
12 DSPEND @ DSPLST C@ OR
13 DBM LSTRST ;
14
15 ==>

```

Screen: 43

```

0 ( DBREM DBDELL DBMODL DBREML )
1
2 : DBREM ( n -- )
3 DUP 1+ LSTSV
4 SWAP DBPTR
5 DSPEND @ DSPLST C@
6 15 AND DBM LSTRST ;
7
8 : DBDELL ( -- )
9 DSPBLK @ 1- DBPTR ;
10
11 : DBMODL ( -- )
12 DSPBLK @ 1- DBMOD ;
13
14 : DBREML ( -- )
15 DSPBLK @ 1- DBREM ; -->

```

Screen: 44

```

0 ( Graph Sys: ?DBVAL )
1
2 : ?DBVAL ( n -- )
3 DSPBLK C@ @ JMPDAT C@
4 ROT DBPTR
5 DSPEND @ DSPLST C@
6 <ROT @ JMPDAT C!
7 DBPTR ;
8
9
10
11
12
13
14
15 ==>

```

Screen: 45

```

0 ( Graph Sys: ?ANTMOD ?DBMODS )
1
2 : ?ANTMOD ( n -- )
3 ?DBVAL DUP 15 AND 0=
4 IF 127 ELSE 15 ENDIF
5 AND ;
6
7 : ?DBMODS ( n -- )
8 ?DBVAL DUP 15 AND 0=
9 IF
10 DROP 0
11 ELSE
12 240 AND
13 ENDIF ;
14
15 -->

```

Screen: 46

```

0 ( Graph Sys: DBMS DBDELS )
1
2 : DBMS ( # n -- )
3 SWAP 0
4 DO
5 DUP DBM
6 LOOP
7 DROP ;
8
9 : DBDELS ( # n -- )
10 SWAP 0
11 DO
12 DUP DBDEL
13 LOOP
14 DROP ;
15 ==>

```

Screen: 47

```

0 ( Graph Sys: DBREMS )
1
2 : DBREMS ( # n -- )
3 SWAP 0
4 DO
5 DUP DBREM 1+
6 LOOP
7 DROP ;
8
9
10
11
12
13
14
15 -->

```

Screen: 48

```

0 ( Graph Sys: DBINS )
1
2 : DBINS ( # n n -- )
3 ROT DUP DBCNT !
4 6 + SDTMP @ SWAP SDTMP !
5 (ROT DUP LSTSV
6 SWAP DBPTR SWAP
7 BEGIN
8 DBCNT @ ( count zero? )
9 WHILE
10 DUP DBM ( keep creating)
11 -1 DBCNT +! ( dec. counter )
12 REPEAT
13 DROP LSTRST ( unsave list )
14 SDTMP ! ;
15 ==>

```

Screen: 49

```

0 ( Graph Sys: SCRVID LSTMV )
1
2 : SCRVID ( width -- )
3 559 C@ 252 AND
4 OR 559 C! ;
5
6 : LSTMV ( -- )
7 LSTLOC @ DSPEND @
8 DSPLST 65
9 OVER C! 1+ !
10 0 DSPLST LSTLOC @
11 DSPEND @ CMOVE ;
12
13
14
15 -->

```

Screen: 50

```

0 ( Graph Sys: USRDSP MIXED )
1
2 : USRDSP ( -- )
3 LSTMV
4 559 C@ 3 AND
5 0 SCRVID
6 LSTLOC @ 560 !
7 SCRVID ;
8
9 : MIXED ( -- )
10 DMLOC @ 88 !
11 USRDSP ;
12
13
14
15 ==>

```

Screen: 51

```

0 ( Graph Sys: DMPLST )
1
2 : DMPLST ( -- )
3 CR DSPBLK @ -DUP
4 IF
5 CR ." BLK ADDR"
6 ." BYTE MODS"
7 CR ." --- ----"
8 ." ---- ----"
9 0 DO
10 CR I 3 .R
11 I DBLST @ 9 U.R
12 I ?ANTMOD 8 U.R
13 I ?DBMODS -DUP
14 IF
15 3 SPACES -->

```

Screen: 52

```

0 ( Graph Sys: DMPLST )
1
2 DUP 128 AND
3 IF ." I" ENDIF
4 DUP 32 AND
5 IF ." V" ENDIF
6 DUP 16 AND
7 IF ." H" ENDIF
8 64 AND
9 IF ." J " I DBLST @ U.
10 ENDIF
11 ENDIF ?EXIT
12 LOOP
13 ELSE
14 ." No display list"
15 ENDIF CR ; ==>

```

Screen: 53

```

0 ( Graph Sys: DMCHG DMCLR DBADR )
1
2 : DMCLR ( -- )
3 DMLOC @
4 106 @ 256 *
5 OVER -
6 ERASE ;
7
8 : DBADR ( blk# -- a )
9 DBLST @ ;
10
11
12
13
14
15 -->

```

Screen: 54

```
0 ( VAL-FORTH GRAPHIC SYSTEM 1.1 )
1
2 : DBWID          ( width -- )
3   0 LSTSV SWAP BLKNML
4   [ 0 BYTBK ] LITERAL
5   16 CMOVE
6   BLKOF5 @ [ 0 BYTBK 16
7   OVER + ] LITERAL LITERAL
8   DO
9     I C@ DUP 3 PICK / + I C!
10  1 /LOOP
11  HSOFS C@ HS# / !
12  LSTRST ;
13
14 '( PERMANENT PERMANENT )( )
15 BASE !
```

Screen: 57

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 55

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 58

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 56

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 59

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 60

```
0 ( Transients:  setup          )
1 BASE @ DCX
2
3 HERE
4
5
6 741 @ 4000 - DP !
7 ( SUGGESTED PLACEMENT OF TAREA )
8
9
10 HERE CONSTANT TAREA
11   0 VARIABLE TP
12   1 VARIABLE TPFLAG
13   VARIABLE OLDDP
14
15                               ==>
```

Screen: 63

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 61

```
0 ( Xsients: TRANSIENT PERMANENT )
1
2 : TRANSIENT          ( -- )
3   TPFLAG @ NOT
4   IF HERE OLDDP ! TP @ DP !
5     1 TPFLAG !
6   ENDIF ;
7
8 : PERMANENT          ( -- )
9   TPFLAG @
10  IF HERE TP ! OLDDP @ DP !
11    0 TPFLAG !
12  ENDIF ;
13
14
15                               -->
```

Screen: 64

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 62

```
0 ( Transients: DISPOSE          )
1 : DISPOSE  PERMANENT
2   CR ." Disposing..." VOC-LINK
3   BEGIN DUP   0 53279 C!
4   BEGIN @ DUP TAREA U<
5   UNTIL DUP ROT ! DUP 0=
6   UNTIL DROP VOC-LINK @
7   BEGIN DUP 4 -
8   BEGIN DUP   0 53279 C!
9   BEGIN PFA LFA @ DUP TAREA U<
10  UNTIL
11    DUP ROT PFA LFA ! DUP 0=
12  UNTIL DROP @ DUP 0=
13  UNTIL DROP [COMPILE] FORTH
14  DEFINITIONS ." Done" CR ;
15  PERMANENT      BASE !
```

Screen: 65

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```


Screens 66 thru 89 are blank

Screen: 90

```

0 ( Utils: CARRAY ARRAY )
1 BASE @ HEX
2 : CARRAY ( cccc, n -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ALLOT
5 ;CODE CA C, CA C, 18 C,
6 AS C, W C, 69 C, 02 C, 95 C,
7 00 C, 98 C, 65 C, W 1+ C,
8 95 C, 01 C, 4C C,
9 ' + ( CFA @ ) , C;
10
11 : ARRAY ( cccc, n -- )
12 CREATE SMUDGE ( cccc: n -- a )
13 2* ALLOT
14 ;CODE 16 C, 00 C, 36 C, 01 C,
15 4C C, ' CARRAY 08 + , C; ==>

```

Screen: 91

```

0 ( Utils: CTABLE TABLE )
1
2 : CTABLE ( cccc, -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 ;CODE
5 4C C, ' CARRAY 08 + , C;
6
7 : TABLE ( cccc, -- )
8 CREATE SMUDGE ( cccc: n -- a )
9 ;CODE
10 4C C, ' ARRAY 0A + , C;
11
12
13
14
15 -->

```

Screen: 92

```

0 ( Utils: 2CARRAY 2ARRAY )
1
2 : 2CARRAY ( cccc, n n -- )
3 (BUILDS ( cccc: n n -- a )
4 SWAP DUP , * ALLOT
5 DOES)
6 DUP >R @ * + R) + 2+ ;
7
8 : 2ARRAY ( cccc, n n -- )
9 (BUILDS ( cccc: n n -- a )
10 SWAP DUP , * 2* ALLOT
11 DOES)
12 DUP >R @ * + 2* R) + 2+ ;
13
14
15 ==>

```

Screen: 93

```

0 ( Utils: XC! X! )
1
2 : XC! ( n0...nm cnt addr -- )
3 OVER 1- + >R 0
4 DO J I - C!
5 LOOP R) DROP ;
6
7 : X! ( n0...nm cnt addr -- )
8 OVER 1- 2* + >R 0
9 DO J I 2* - !
10 LOOP R) DROP ;
11
12 ( Caution: Remember limitation
13 ( on stack size of 30 values
14 ( because of OS conflict. )
15 -->

```

Screen: 94

```

0 ( Utils: CVECTOR VECTOR )
1
2 : CVECTOR ( cccc, cnt -- )
3 CREATE SMUDGE ( cccc: n -- a )
4 HERE OVER ALLOT XC!
5 ;CODE
6 4C C, ' CARRAY 08 + , C;
7
8 : VECTOR ( cccc, cnt -- )
9 CREATE SMUDGE ( cccc: n -- a )
10 HERE OVER 2* ALLOT X!
11 ;CODE
12 4C C, ' ARRAY 0A + , C;
13
14
15 BASE !

```

Screen: 95

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screens 96 thru 167 are blank

Screen: 168

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 171

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 169

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 172

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 170

0 CONTENTS OF THIS DISK:
1
2 DISPLAY FORMATTER: 30 LOAD
3 TRANSIENTS: 60 LOAD
4 ARRAYS & THEIR COUSINS: 90 LOAD
5
6
7
8
9
10
11
12
13
14
15

Screen: 173

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages)
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 179

0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

TURTLE & valGRAPHICS^{T.M.}
AND ADVANCED
FLOATING POINT ROUTINES

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

Software and Documentation
© Copyright 1982
Valpar International

valFORTH^{T.M.}
SOFTWARE SYSTEM

TURTLE & valGRAPHICS^{T.M.}
AND ADVANCED
FLOATING POINT ROUTINES

DRAWLN ROUTINES: Stephen Maguire
ARMADILLO GRAPHICS: Evan Rosen, William Volk
QUAN STRUCTURES: Evan Rosen

Software and Documentation
©Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

valGRAPHICS 1.2 AND QTRIG USER'S MANUAL

Table of Contents

	<u>Page</u>
XLI. valGRAPHICS	
a) Overview	2
b) Strolling Through valGraphics (Walking the Armadillo)	5
c) Strolling Through valGraphics, II (Windows, Lines, and Labeling)	14
XLII. valGRAPHICS GLOSSARIES	
a) General Functions	1
b) Windows and Coordinate Systems	7
c) Options	10
d) Screen Dump	14
e) Interfacing to Custom Display Lists	15
f) A Note on QUAN Structures	16
XLIII. QUICK TRIG	
XLIV. valGRAPHICS SUPPLIED SOURCE LISTING	

valGRAPHICSTM

Version 1.2
July 1982

Many different approaches to creating "computer graphics" are possible, and, indeed, many have been implemented. One of the most fruitful approaches, particularly for two-dimensional graphics work, is a system usually called "turtle graphics." The valGraphics package is a turtle-like system patterned after the ATARI PILOT turtle graphics rendition, though with many significant extensions.

Software and Documentation
© Copyright 1982
Valpar International

Overview

Turtle geometry was originally developed at MIT by Dr. Seymour Papert and the LOGO group there. Since that time, a variety of different computer-based applications have been said to support "turtle graphics," though in general they differ in various ways from the initial LOGO formulation. The formulation adopted for this package follows the ATARI PILOT turtle graphics nomenclature where possible. In general, commands in this package are much smarter than their PILOT counterparts, but have also been made friendly so that in their default modes they are functionally similar to PILOT commands. Because this package differs from "true" turtle graphics in many ways, it is called "armadillo graphics."

Some minor changes have been made in the names of a few ATARI PILOT commands because of collisions with existing valFORTH names. Important variations are listed here:

<u>PILOT</u>	<u>valGraphics</u>
CLEAR	WIPE (CLEAR exists)
FILL	PHIL (FILL exists)
FILLTO	PHILTO (for consistency)
QUIT	(not needed and exists)
PEN ERASE	O PEN (ERASE exists)
LOCATE	LOOK (LOCATE exists)

It should be mentioned that for this package, virtually the entire set of ATARI operating system graphics functions have been replaced by much faster (approximately 6 times) and much smarter graphic routines. Highlights of these improvements are:

- * The PHIL and PHILTO functions allow filling to the left and/or right, filling across areas already filled, filling until a specific color is hit, filling until a specific color is not hit (i.e., re-filling), filling until hitting a set boundary regardless of what lies in the way, filling by either replacement or exclusive OR'ing, and filling into or out of corners without artifacting.
- * The DRAW and DRAWTO functions allow drawing by either replacement or exclusive OR'ing, drawing until hitting a specified color, drawing until a specific color is not hit. Additionally, lines are more symmetric, and optional starting point plot is supported (the Atari routines never plot the first point of a line).
- * All line drawing and fill routines allow plotting in wide and narrow screen width settings as well as normal ones, allowing true full screen graphics and memory conservative graphics, for the advanced programmer.

- * Graphic mode 7+ ("7 and a half") is fully supported and is activated like any other graphic mode by using the GR. command and standard options. This mode is the four-color high-resolution graphic mode found in most of the better video games available for the Atari computers. (Antic mode 14.)
- * Display windowing and clipping is full supported. Options include "machine specific" coordinates for speed and "scaled" coordinates for portability.
- * TURNTWD (turn toward) and 2LNK (two line intersection) commands are available allowing simplified perspective drawing.

Although the Atari 400/800 computers have extensive graphic capabilities the need to keep the ROM operating system under 10K apparently forced Atari to omit the highest resolution color graphic mode (graphic mode 7+) and allowed only inefficient draw and fill routines to be implemented. Since this severely limited the usefulness of the computer for plotting, both of these problems have been corrected.

A New Graphic Mode

Many of the better video games for the Atari 400/800 computers use a color graphic mode not supported in BASIC. By redefining the GR. command, it was possible to implement this previously unavailable mode.

This new graphic mode, which in this package is called graphic mode 12, is similar to graphic mode seven. The difference is that a pixel (a single dot) in graphic 12 is half as tall as the same pixel in graphic mode 7. This mode is activated in the same manner as other graphic modes:

12 GR.

All options (i.e., split/full screen etc.) available for other modes will work with this new mode. In the split screen graphics 12 mode, there are 160 horizontal by 160 vertical pixel locations. In the full screen mode, there are 160 horizontal by 192 vertical locations. Note that to use this mode the valGraphics package must be loaded and the new draw routines must be used (the operating system routines fail in this mode).

Draw and Fill Routines

Because the line and fill routines in this package represent significant enhancement to the original operating system routines, an explanation of the why's and how's of this implementation is offered in the following.

It was first decided that the line-drawing routines must be speeded up so as to at least be in the class of routines of other 8 bit graphic machines. Because of the differing bit structures in the various graphics modes, these routines take up about 1000 bytes of memory. This was deemed a reasonable tradeoff. Since a complete rewrite had thus been elected, the opportunity was taken to expand the versatility of the routines, trading a small portion of the speed increase already gained. Several capabilities were deemed desirable and were implemented:

- * As mentioned above, the draw routines work in graphics mode 12.
- * Assuming that display memory has been properly laid out, the draw routines work in wide and narrow screen widths as well as the normal ones.
- * The draw and fill functions, at user option, XOR rather than replace pixels in display memory so that new images can be written over background images. (Images are then erased by rewriting, restoring the background image.)
- * The draw and fill functions can detect a variety of conditions so as to allow concepts like "draw until" and "draw until not" as well as "fill until" and "fill until not."
- * The fill function allows the edge color and the surface color to be different, at user option, with the default setting that they are the same.
- * The fill function allows filling to the left, right, or both simultaneously, at user option.
- * Fills are able to start from and pass through corners without artifacting, at user option. (Implemented for vertical draw only.)
- * Simple initialization of draw functions for custom display lists is provided.

These features were implemented and will be described shortly.

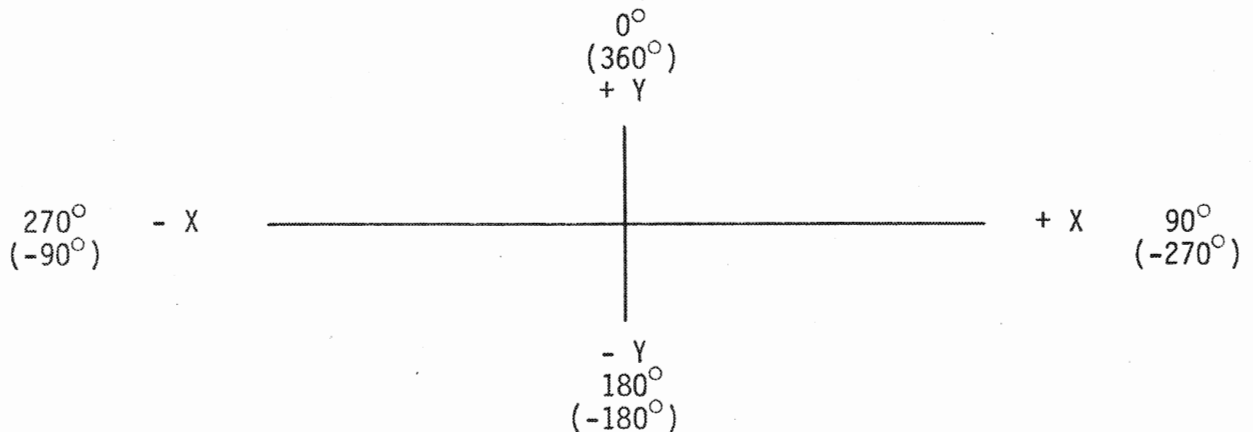
STROLLING THROUGH valGRAPHICS

Walking the Armadillo

To get started, insert your valGraphics disk and load in the armadillo package and all optional graphics packages ("+'s") (including the demos). It would even be a good idea to SAVE a copy of this system in case you crash later on. (Insert a formatted disk and type SAVE.) The load addresses may be found on screen 170 of the disk. Note that you do not need, and probably won't want, to first load the graphics package provided on your valFORTH 1.1 disk. Note also that as these packages load, some load comments may be reported as "xxxx Is not unique" and can be ignored. This message simply states that a word has just been defined with the same name as an already existing word.

When plotting in BASIC, location (0,0) is in the upper lefthand corner of the video display. All horizontal and vertical positions to the right and down are referenced with positive offsets from the (0,0) point. Armadillo graphics uses a somewhat different method to specify a location.

In armadillo graphics, the point (0,0) is located in the center of the display. Horizontal locations to right are referenced with positive offsets from this point, while locations to the left are referenced using negative offsets. Likewise, locations higher on the screen from the origin are referenced with positive vertical offsets while those lower on the screen are referenced using negative ones. Since this setup follows the standard cartesian coordinate system, function plotting is greatly simplified.



Let's take a look at the basic armadillo graphic commands. Type:

8 GR. ON ASPECT

This will put the system into graphic mode 8 with the armadillo positioned in the center of the display facing upward (0 degrees). The "dimensions" of the display are 320 pixels wide and 160 pixels high. The boundaries are set from -159 to 159 left-to-right, and 79 to -79 top-to-bottom. (The lowest line of pixels and the furthest right are excluded for code symmetry and shortness.) The command "ON ASPECT" will be explained later, but basically it ensures that squares will look like squares and not like rectangles (as in BASIC). First let's turn the armadillo to the right. To change its direction, we use the TURN command:

90 TURN

This command turns the armadillo clockwise by 90 degrees from its current direction. To draw a line (and move) the armadillo, the DRAW command can be used. Try this:

30 DRAW 90 TURN 50 DRAW

A short line should have been drawn toward the right -- 30 steps in the direction the armadillo was facing. The 90 TURN command was then used to aim the armadillo downward, and 50 steps in that direction were taken. The DRAW command moves the armadillo the specified number of steps in the direction that it is facing. Note that a negative step count tells the armadillo to draw in the direction opposite that in which it faces. It is also possible to move the armadillo to a specified point on the screen regardless of which direction it is facing. The DRAWTO command is used for this:

0 -60 DRAWTO 0 TURNT0

Although the armadillo was facing down, it moved directly to the point (0,-60). Note that although it moved diagonally, it still is facing directly downward (to 180 degrees). The TURNT0 command is used to face the armadillo in the specified direction regardless of where it is currently facing. In this case, the armadillo is turned to face 0 degrees.

In addition to drawing lines as it moves, the armadillo can fill in areas of the display. The PHIL command is used for this purpose and functions very much like the DRAW command. (FILL is already defined and if used mistakenly for PHIL, the system will probably crash.) Try this:

20 PHIL

This commands the armadillo to take 20 steps in its current direction filling the surface area to its right as it goes (the area to the left can be filled also -- more on that later). Similar to the DRAWTO command, there is also a PHILTO command which works just like PHIL except that the armadillo moves to a specified point regardless of the direction it is currently facing. To PHIL to the origin (0,0), use:

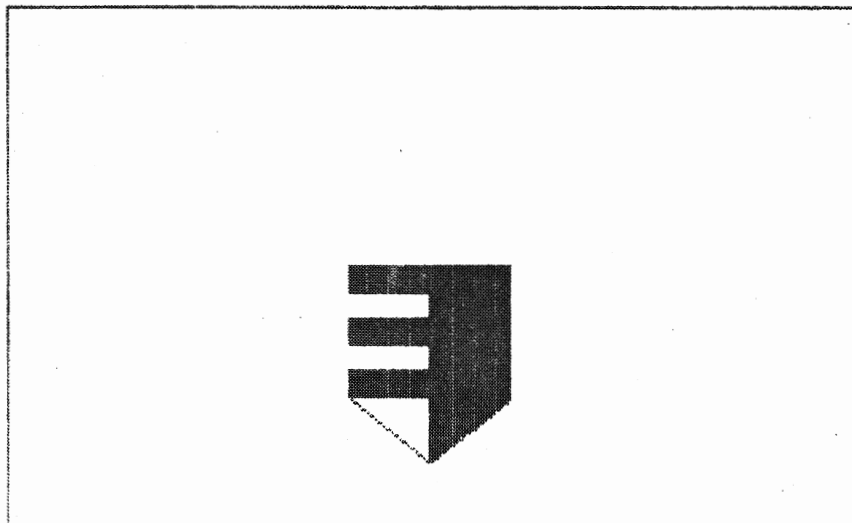
0 0 PHILTO

The PHILTO command should have filled straight up to the point (0,0).

So far, we have used the TURN, TURNT0, DRAW, DRAWTO, PHIL, and PHILTO commands. These are the basic "drawing" words, used constantly, when working with armadillo graphics. You will encounter times when you need to move the armadillo without drawing a line between its starting point and its destination point. There are four similar commands which allow this. The GO, GOTO, GO., and GOTO. All reposition the armadillo without drawing a line. The GO and GOTO commands function like DRAW and DRAWTO respectively; however, the armadillo is placed at the position where the last dot of the plotted line would have been and no line is drawn. GO. and GOTO. function like GO and GOTO; however, a single point is plotted at the destination point. Try this:

```
-30 0 GOTO.      180 TURNT0
10 PHIL      10 GO      10 PHIL      10 GO      10 PHIL
0 -60 DRAWTO
```

After entering the above, type FRAME to frame this picture. If all went well, your display should look like:



Let us now explore the new graphic 12 mode. In this mode, there are four colors numbered zero to three. When the armadillo is moved, there must be some way to specify which color to DRAW with. The PEN command is used for this purpose. Enter the graphic 12 mode by typing:

12 GR. ON ASPECT

The GR. command automatically sets the draw color to one (usually red). Let's draw some colored lines now:

```
10 DRAW                      (draw in color 1)
2  PEN          10 DRAW      (draw in color 2)
3  PEN          10 DRAW      (draw in color 3)
```

You should now have a vertical red, green, and blue line. Note that color 0 is black (actually background) and is used primarily for erasing lines. Besides setting the draw color, the PEN command also sets the PHIL color.

```
-50 0 GOTO    45 TURN
2  PEN      10 PHIL
```

After positioning the armadillo in a good position for filling, the draw and fill color is set to 2 (usually green) and 10 steps are taken. By using PHPEN command, it is possible to set the PHIL color to something other than the PEN color. PHPEN stands for "phil-pen" and is used in the same manner as the PEN command:

```
3  PHPEN    32 PHIL
```

Note how the edge line remains the color set by the last PEN command, while the PHIL command uses the color set by the last PEN or PHPEN (whichever came last) command. The PHIL color is always set by the PEN command for convenience. Experiment with this a bit.

To clean the current display, the WIPE command is used. Usually after wiping the display, the armadillo is repositioned to the center of the screen using either the CENTER or CENTER0 command. The CENTER command simply does a "0 0 GOTO" while the CENTER0 command does a "CENTER 0 TURNTO".

WIPE CENTER0

Either right or left filling can be performed, as well as both simultaneously. The two commands RPHIL and LPHIL take an ON/OFF value and instruct the next PHIL or PHILTO command to take appropriate action. The default setting is "ON RPHIL" and "OFF LPHIL". The command DINIT will return all settings to their default values. This is especially valuable when learning, as it is easy to get fouled up. Type in the following set of commands and observe what happens:

```
50 50 PHILTO
CENTER OFF RPHIL ON LPHIL
-50 50 PHILTO
CENTER ON RPHIL 1 PEN
40 PHIL
```

This demonstration first shows right filling, then left filling, and then simultaneous right/left filling. Note that although there was nothing on the screen to halt the right/left fills, they still stopped upon hitting the edge of the display. In fact, by using windows (which will be described later), fill boundaries can be set anywhere on the screen and fills will never occur outside of those boundaries. This is invaluable when trying to restrict drawing and filling to a select portion of the display.

Another unique feature of the fill routines is that they allow filling over any pseudo-background color (default is 0). The PHBAK command is used to specify this background color. Like the PEN and PHPEN commands, PHBAK accepts a color specification on the stack. WIPE uses the color specified by the last PHBAK command, and the fill routines recognize this as background to be filled over. Try this:

```
3 PHBAK    WIPE
CENTERO    0 PEN    1 PHPEN
50 50 PHILTO
```

For the time being, we will leave the background color blue and continue on. Next we are going to define a few words which will draw simple shapes. Bear in mind that when defining shape words, TURNT0, DRAWTO, and PHILTO should be avoided as they are absolute in nature. Typically, figures should be drawn relative to the armadillo's direction. Likewise, the armadillo should generally be returned to its original position and heading once the "canned" shape has been drawn. (For the curious, the words DX1 and DY1 return the x-y coordinates of the armadillo's current location. The word DAZM returns the directional angle of the armadillo.) We shall now define a word which will draw a square on the screen.

```
: SQUARE                                ( #steps/side --- )
  DUPDRAW    90 TURN
  DUPDRAW    90 TURN
  DUPDRAW    90 TURN
  DRAW       90 TURN ;

WIPE  CENTERO  2  PEN
20  SQUARE
45  TURN      20 SQUARE
```

There are several points to be mentioned here. First, because combinations of DUP with DRAW, GO, and GO. occur often, the words DUPDRAW, DUPGO, and DUPGO. have been defined to conserve memory.

Also notice that the squares drawn really have sides of equal length (in BASIC, the vertical legs would be much shorter). The armadillo package performs "aspect ratio" calculations which ensures that "equal" lines are drawn the same length regardless of their orientation to a fixed axis. These routines were enabled at the beginning of this stroll with the "ON ASPECT" command. Because these calculations do take time (approximately 3 milliseconds per draw), they can be turned off using the command:

```
OFF ASPECT
```

Now, notice how in the last example, the second square was rotated 45 degrees from the first square. We can write simple words using this effect that look pretty snappy on the screen:

```

: FAN                                ( #steps/side --- )
  20 0                                ( 20 squares for fan )
  DO
    18 TURN                          ( 360 degrees/20 = 18 )
    DUP SQUARE
  LOOP
  DROP ;

WIPE  CENTER  30 FAN

```

This word draws 20 squares on the screen each offset from each other by 18 degrees. Try changing PEN colors and give different step sizes to FAN and watch the results. Each of the boxes drawn by FAN is the same size. We can write another simple word which will slightly increase the size of each box drawn and obtain a different effect:

```

: WHIRL                              ( #boxes --- )
  ( #boxes ) 0
  DO
    I 3 / SQUARE                    ( increase size )
    5 TURN
  LOOP ;
WIPE  CENTER  250 WHIRL

```

This word draws the specified number of boxes, each one rotated from the last by 5 degrees. After three boxes are drawn, the box size is increased. This is how the swirl effect is obtained. A slight variation of this is to change the PEN color before each square is drawn, but this is left to the reader.

Up to now, we have drawn lines from one point to another regardless of what the line replaces. This is standard for line drawing routines. In the valGraphics package, however, "draw until" is supported. In other words, lines can be drawn that will stop on the first occurrence of another line (actually, until the color specified by PHBAK or DRBAK, whichever came last). When the draw-until switch DRUNT is ON, all DRAW, DRAWTO, PHIL, and PHILTO commands will stop when the base line hits another line on the display. Here's an example:

```

ON DRUNT  WIPE  CENTERO
40 SQUARE

```

Don't worry if only the two vertical sides of the square were drawn, this is normal. Since the draw routines in this package plot both the end point and the starting point, the end point of the first side stopped the line draw of the second side. In most cases, this is the desired function for DRAW, but while drawing-until (ON DRUNT), first point plotting is not desired. For this reason, it can be easily turned off using the DR1ST switch:

```

OFF DR1ST  WIPE
40 SQUARE

```

With DRUNT still on, try the following example:

```
WIPE      30  QCIRCLE
40  SQUARE
```

The QCIRCLE command draws a quick-and-dirty circle about the armadillo, with the value on top of stack taken as the approximate radius. Notice how the 40-step square turned out. Because the draw-until mode is on, each side is drawn until another line is encountered. Note, however, that even though the line was not drawn to the destination point, the armadillo was still positioned there. Because interesting results can be obtained by using this feature, the RELOC command must be used explicitly to reposition the armadillo to the last plotted point of the line. The following two definitions might come in handy:

```
: DRWUNTIL                      ( #steps --- )
  DILLO
  ON DRUNT
  DRAW DFLG
  IF RELOC ENDIF
  OFF DRUNT ;

: DRW2UNTIL                      ( x y --- )
  DILLO
  ON DRUNT
  DRAWTO DFLG
  IF RELOC ENDIF
  OFF DRUNT :
```

(DFLG is a flag set true only if the last DRAW or DRAWTO crossed the current window.)

These two commands will automatically reposition the armadillo at the end of the drawn line after each draw. One last point about draw-until -- occasionally it is desirable to know when a draw-until line was stopped by the draw-until function, rather than by reaching an end point or window boundary. The ?DRSTP word will return a one (1) if the last line was stopped, otherwise it will return zero. Try drawing a few lines and verifying this. ?DRSTP is in the DILLO vocabulary. (See the glossary.)

Up until now, when we filled areas or drew-until, both the fills and draws would stop when encountering a non-pseudo background color (set by PHBAK or DRBAK). Often, it is desirable to refill an area (i.e., fill until background is hit) or draw-until hitting the pseudo background color. There are two switches which can be turned ON or OFF as desired. The PHUNOT (fill until not) switch, when ON, fills until the color set by the last PHBAK command is not hit. This is the default condition (i.e., fill until background is not hit). When OFF, the fill routines continue to fill until the pseudo background color is hit. Likewise, the DRUNOT (draw until not) switch, when ON, draws until the color set by the last PHBAK or DRBAK (whichever came last) command is hit. Let's take a look at this:

```
DINIT                      (Reset draw/fill switches)
2  PEN      3  PHPEN
WIPE  CENTERO  50  50  PHILTO
1  PHPEN  180  TURNT0
OFF  PHUNOT                      (fill while not background)
OFF  RPHIL      ON  LPHIL  60  PHIL
```

As you may recall, the DINIT command initializes all eleven switch settings (five of which have yet to be introduced). Next a normal right fill (filling over background) is performed. The PHUNOT switch is then set for filling while not background and a left fill is performed. Notice that no filling occurred when the base fill-line extended out of the previously filled area.

DRUNOT works in the same manner. Execute the last example a second time, but turn both PHUNOT and DRUNOT off where previously just PHUNOT was turned off. Also turn DRUNT on. This time, no line should extend past the previously filled area. (Note that the base line of a fill responds to all the draw switches).

To finish off this first part of the stroll, the final five draw switches will be explained. Briefly, they are PH-DR which allows the base line of a fill to be drawn or not, DRXOR and PHXOR which allow lines and fills to be XOR'd into place, PHCRNR which enables/disables rudimentary corner check tests for filling, and PHUNT which allows filling to the edge regardless of what lies in the way.

The PH+DR switch is available because there are times when it is not desirable to actually draw the base line of a fill. This is the case when PAINTing (i.e., "shape filling," which is not supported but may be implemented). The default value for PH+DR is ON.. When PH+DR is OFF, the pixels where the base line should be drawn are left untouched.

The DRXOR and PHXOR switches allow lines and fills to be XOR'd into place. This has the useful property that by simply redrawing or refilling the exact same line or shape the object will erase itself. For a good example of this, we can use graphic mode 8:

```
8 GR. DINIT 250 WHIRL
1 PHBAK      ON DRXOR   WIPE
```

Recall that the WIPE command uses the value set by the PHBAK command—in this case, one. WIPE is defined to use a multiple DRAW and therefore responds to most (but not all) of the draw switches. Because the WIPE is performed with the DRXOR mode on, the display is inverted. WIPE the display a second time to re-invert it. To erase the display, DRXOR must be turned off. Try this:

```
0 PHBAK      OFF DRXOR   WIPE      ON      DRXOR
1 PEN        CENTERO    40 FAN
40 FAN                                     ( one more time )
```

It is important to remember that lines drawn with pen zero have no effect in the DRXOR mode. Likewise, first point plot should generally be turned off when DRXOR is on otherwise endpoints will be lost. Now to demonstrate PHXOR and PHUNT try these examples:

```
DINIT          WIPE          (normal situation)
CENTERO        100 WHIRL      50 50 PHILTO
```

Now with PHUNT off:

```
CENTERO      OFF PHUNT      50 50 PHILTO
```

The last fill command should have filled clear to the edge of the display, ignoring everything in its path. Using this with PHXOR, interesting results can be obtained:

```
WIPE      CENTERO      50 FAN  
ON PHXOR      50 50 PHILTO
```

Now, try:

```
CENTERO      50 50 PHILTO
```

By using windows (described later), the fill and draw commands can be restricted to selected areas of the display. In combination with windows, PHXOR can produce astounding visual effects (especially in GTIA modes).

The last remaining switch to be described is the PHCRNR switch. PHCRNR allows rudimentary corner checking for vertical fills. Because its use is specialized, PHCRNR is normally turned off. The following example will show its function:

```
DINIT      CENTERO      WIPE  
50 GO      50 0 DRAWTO      0 0 DRAWTO      50 PHIL
```

Notice the artifact at the top corner. Now, turn PHCRNR on and perform the same example (less the DINIT command). This time, no artifact should have appeared. It is important to remember that these corner tests will not work with many diagonal fills, and completely fail when refilling an area. Also note that when first point plot is disabled, even vertical filling fails.

All of the basic armadillo commands have been explained and are summarized in the glossary and on the valGraphic Handy Reference Card. Although many commands have been discussed, there are many more left to talk about. These include the perspective drawing commands for three dimensional displays, and the complete set of window commands which will be described next.

(NOTE: In all of the above examples, WIPE has been used to clear the display. In many cases, the memory FILL command can be used instead: 88 @ n 0 FILL where n is the size of display memory in bytes. This method is much faster but cannot be used with windows.)

STROLLING THROUGH valGRAPHICS, PART II:

Windows, Lines and Labeling

Windows

Up until now we've been working in the base window that is set up when using the GR. command. Let's compose some other windows. Type:

```
12 GR.  
FRAME  
10 QUBE  
-50 -10 30 -10 WINDOW  
FRAME  
DOT
```

We entered graphics 12, framed the base window, made a window whose left, right, top, and bottom edges were at -50, -10, 30, and -10 respectively, framed it, and then put a dot at the armadillo and found that it was at the center of the new window. Now type

```
25 QUBE
```

and note that the cube is clipped within the boundaries of the new window, not the old one. This could be very useful, say, in showing what was visible through a "real" window in a house that you had drawn, without going to a lot of extra trouble to restrict the image to the house's window. Now type

```
WIPE  
CENTERO  
25 QUBE
```

Nothing happens. This is because CENTERO centered the armadillo in the base window. We need to use a different word to re-center in the new window. Type

```
WCTRO 25 QUBE
```

That's more like it. WCTRO stands for "Window CenTeR O turnto," and there is also just a WCTR, for "Window CenTeR." Let's try some of the other tricks from before:

```
3 PHBAK  
WIPE  
WCTR  
20 QUBE  
FRAME  
ON DRXOR  
WIPE
```

Get the idea? When we did ON DRXOR, the draw routines, which are used by WIPE, started doing an XOR instead of a replace, with the same effect as we've seen before, but this time restricted to a smaller window. Type

DINIT
WIPE
DOT

to get things back to normal. Note that DINIT returns to the base window and so WIPE wipes the entire screen. The window in which we were just working is forgotten. (We'll discuss ways to remember it a little later.) DOT shows that the armadillo is back at the center. Now type

ON ASPECT
-50 -10 30 -10 WINDOW
FRAME

and you see that this "same" 40 by 40 window as before now looks much more nearly square. This illustrates that ASPECT works on windows as well as lines. With ASPECT on, what you give up in order to get better shapes is some information about what coordinates the top and bottom of the screen actually are, but for "hands on" use this is not much of a loss. Let's make two more of this type of window:

2 PEN
0 30 20 -30 WINDOW FRAME
DOT

OK, and then

-30 30 90 -90 WINDOW

Notice that this window is larger than the base window. Now type

DOT
FRAME

and notice the trash in the text window. If you choose to make a window larger than the base window, the system will not protect you; it assumes that you know what you're doing.

Type CLS once or twice to clear the screen. Then type

12 GR.
10 50 -10 -60 RELWND
FRAME
2 PEN
40 QUBE

Interesting. Now a 40 QUBE used to be much bigger; but because we typed RELWND instead of WINDOW, objects are drawn relative to the new window, as if it were the base window. Type

3 PHBAK
WIPE
FRAME
10 50 -10 -60 RELWND
FRAME
40 QUBE

Get the idea? Relative windows are useful for all sorts of tricks. Often, it would be helpful to be able to return to a window, and relative windows are the hardest to reconstruct. Try typing, on one line,

```
THISWND LIVING-ROOM
```

(Defining words should always be followed by the name of the new word on the same line.)

By typing LIVING-ROOM later on we can return to this window, as a relative window, with no further work. To demonstrate, type

```
0 PHBAK
WIPE
BASWND
ON DRXOR
1 PHBAK
WIPE
-20 0 20 0 WINDOW WIPE
THISWND MY-ROOM
LIVING-ROOM
WIPE
50 QUBE
MY-ROOM
50 QUBE
```

Normal windows, created by WINDOW, of course can also be named more directly:

```
: window-name number number number WINDOW ;
```

and you've got it.

Well, what else? Type

```
40 GR.
```

What have we here? 40 is $32 + 8$ so we've entered 8 GR. without pre-erasing. (This is one of the standard GR. options, you'll recall.) Since 12 GR. and 8 GR. occupy exactly the same display memory, what we see is the 12 GR. image data interpreted as 8 GR. Four color 8 GR. This effect has been written up in various places, and here it is. You can come back and play with this sometime. Right now, type

```
DINIT
WIPE
DOT (you may not be able to see it on your screen
    without adjustment.)
0 -30 GOTO 60 DRAW
30 0 GOTO 270 TURNTOW 60 DRAW
(Now it's more visible)
87 -31 GOTO.
(Move away)
0 0 TURNTOW ("turn-toward")
100 DRAW
```

You'll notice that the line doesn't hit 0,0 exactly. This is because the armadillo's direction is only represented to the nearest degree. Still, this is good enough for most purposes.

Finally, let's draw the a rectangular solid in two-point perspective. (The procedure in this example is not necessarily the best one, but it illustrates several capabilities. You might want to have the debugging package loaded from the valFORTH 1.1 disk, and have the stack turned on. That way you can follow the action on the stack also.) First we set up a horizon and two vanishing points:

```
WIPE
-200 60 GOTO 90 TURNTD 500 DRAW
-100 60 NAMEPT VP1 (name the point on stack)
100 60 NAMEPT VP2
```

Then we "construct" the solid

```
CENTER VP1 TURNTWD 40 DRAW
THISPT PT1 (name the present point)
CENTER VP2 TURNTWD 30 DRAW
THISPT PT2
CENTERO 20 DRAW
THISPT PT3
VP1 TURNTWD MAKLN (leave a "line" on the stack)
PT1 GOTO 0 TURNTD MAKLN (leave a second line)
2LN (find their intersection)
NAMEPT PT4 (and name the point)
PT4 DRAWTO PT3 DRAWTO
VP2 TURNTWD MAKLN (do it again)
PT2 GOTO 0 TURNTD MAKLN (second line)
2LN (intersection)
2DUP GOTO (make a copy then go there)
PT3 DRAWTO PT2 GOTO 2DUP DRAWTO (put in 2 more lines)
VP1 TURNTWD MAKLN PT4 GOTO VP2 TURNTWD MAKLN 2LN
DRAWTO DRAWTO (finished)
0 GR. VLIST (see the new words: point names.)
40 GR. (Still there.)
```

In addition to MAKLN there is also THISLN which name the line the armadillo lies on, and NAMELN which will name a line on the stack. Given two points on the stack, 2PT-LN will change the four values into three, suitable for use with NAMELN. Practice, and some study of the glossary, will help. The user should realize that points and lines can't be named very easily within a program, but only while the program is loading. Within a program, use the stack or array structures for saving points and lines.

This stroll is not meant to exhaust the possibilities of this package, but merely to indicate them. A clever programmer, for instance, would have little trouble in figuring out how to interface this package to a joystick to make a very versatile sketchpad.

Hmmmmmm?

(intentionally left blank)

valGRAPHICS GLOSSARIES

Turtle graphics, and so also valGRAPHICS, uses a coordinate system different from that used by the Atari OS. In valGRAPHICS, the center of the graphics display is the point 0,0. "x" values are positive to the right and negative to the left, while "y" values are positive toward the top of the display and negative toward the bottom. The maximum values of x and y may vary between display modes, depending on various user options that are selected. In this regard, see examples in the "Strolling Through valGRAPHICS" sections of this package, and also the words MCOOR and SCOOR in the "Windows" section below. In the glossary that follows, all mention of coordinates will apply to valGRAPHICS coordinates rather than to Atari OS coordinates.

The DRAW, DRAWTO, PHIL, and PHILTO commands support a number of options with a fair degree of complexity and power when used fully. These commands and options are discussed as a group at the end of the glossary and summarized with a chart which also appears on the handy reference card. The functions discussed are necessarily complex; however, the command DINIT ("d-init") is provided so that the user may return the system to a "standard-option" status during experimentation and practice, or during actual program execution.

The term "pixel" stands for "picture element" and refers to the smallest "point" which may be drawn in a given graphics mode.

As usual, "color" specification numbers refer to color registers. The actual colors in the color registers may be changed by various means, including loading the COLOR COMMANDS package from the valFORTH 1.1 disk and using the SETCOLOR or SE. command.

On GTIA-equipped machines in 10 GR. there are nine colors available, because the four player/missile color registers are also used. Since these registers sit just below the playfield color registers in memory, they may be set by using negative "playfield" numbers when using SE. . For instance, -3 PINK 6 SE. will set player/missile 1 (= -3 + 4) to PINK 6.

On GTIA-equipped machines in 9 GR. the "color" set by the various color commands below, e.g. PEN, PHPEN, PHBAK, etc., is interpreted explicitly as luminance between 0 and 15. The hue is that of the background color register.

On GTIA-equipped machines in 11 GR. the "color" set by the various color commands below is interpreted explicitly as a hue between 0 and 15. The lum is that of the background color register.

The term "armadillo" rather than "turtle" will be used in this package.

DILLO (short for armadillo) is a vocabulary that branches from FORTH. All of the system words in this package have been put in the DILLO vocabulary to keep them out of the way during VLIST and other tasks. Some little-used words are also in DILLO, though advanced users may want to get at them. To enter the DILLO vocabulary simply type DILLO and these words will now be recognized by the system. Note that since the word : generally puts the system back into the FORTH vocabulary, DILLO may have to be used within a colon definition. See the source code for numerous examples of this. Words in the DILLO vocabulary are so specified in the glossary below. (The word DILLO is immediate.)

For clarity, some definitions may be repeated. Within this glossary, however, the same name indicates the same word.

valGRAPHICS Glossary
Part I of III

General Functions:

GR. (n --)

Appears to function as always, but is now much more powerful:

* For n = 12, or 12 with higher bits set for the usual options, the mode known popularly as 7+ will be activated. This mode is set by Antic instruction 14 and its characteristics are listed on the handy reference card which accompanies this package.

* For n = 3 to 12, (possibly with higher bit options), the appropriate graphics mode will be set up, and all armadillo parameters will be initialized. Note, of course, that if your machine does not have a GTIA chip, then modes 9, 10, and 11 will not operate as they should.

* For n = 0 to 2, (plus higher bit options), the system will respond as usual.

* GR. initializes a number of system and user quantities. Data about pixel and display-memory dimensions are sent to appropriate addresses. A pen color register of 1 is set by 1 PEN, and the background color register for fill commands is set to 0 by 0 PHBAK. OFF ASPECT is executed.

PEN (n --)

This command is used to change the color that the armadillo draws with. PEN sets a new color register, n, to be used by the DRAW, DRAWTO, PHIL, and PHILTO commands.

PHPEN (n --)

This command is used to change the color that the armadillo fills with. PHPEN sets a new color register, n, to be used by the PHIL and PHILTO commands. Note that PEN also sets a new color register for PHIL and PHILTO, so the value used by PHIL and PHILTO will be determined by whichever command, PEN OR PHPEN, was done last.

DRCLR (-- b)

Returns the present color used by the armadillo for drawing. DRCLR is in the DILLO vocabulary.

PHCLR (-- b)

Returns the present color used for filling. PHCLR is in the DILLO vocabulary.

GO (n --)

GO moves the armadillo n units in the direction in which it is facing. No lines are drawn or points plotted.

DUPGO (n -- n)

Same as GO, but doesn't destroy stack argument.

GO. (n --)

GO. moves the armadillo n units in the direction in which it is facing and then pokes the pixel at its new location with the value set by the last PEN command.

DUPGO. (n -- n)

Same as GO., but doesn't destroy stack argument.

DOT (--)

DOT puts a dot of the present armadillo color, set by PEN, at the present armadillo position.

GOTO (x y --)

GOTO positions the armadillo at x,y. No lines are drawn or points plotted.

GOTO. (x y --)

GOTO. positions the armadillo at x,y and pokes the pixel at the new position according to the color register selected by the last PEN command.

CENTER (--)

Positions the armadillo at the point 0,0. The direction the armadillo is facing is unchanged.

CENTERO (--)

Positions the armadillo at the point 0,0 and turns it to face 0, i.e., straight up.

RELOC (--)

Positions the armadillo at the last point drawn by the system routines. This is a special purpose command and is used in conjunction with clipping in windows, and with the "draw-until" option, described elsewhere. RELOC is in the DILLO vocabulary.

ASPECT (ON or OFF --)

ON ASPECT will cause vertical components of subsequent graphics commands to be scaled to account for the fact that pixels are not square. Thus, circles will be rounder, squares will be squarer, and so on. Of course, shapes that previously fit on the screen may not fit any longer, as a result of the vertical expansion. OFF ASPECT will turn the compensation off for subsequent commands. OFF is the default mode, but this may be altered by changing "OFF ASPECT" to "ON ASPECT" at the end of the source code for GR.

LOOK (x y -- b)

This command returns the value of the pixel at location x,y. LOOK does not move the armadillo. For example, to find the color of the pixel under the armadillo, use the armadillo's coordinates: DX1 DY1 LOOK.

DX1 (-- n)

Returns the x coordinate of the armadillo.

DY1 (-- n)

Returns the y coordinate of the armadillo.

TURN (n --)

Changes the direction that the armadillo is facing by n degrees clockwise. Hence, if n is negative, the armadillo will turn counter-clockwise.

TURNT0 (n --)

Turns the armadillo to a heading of n degrees from vertical. Hence, 0 TURNT0 points the armadillo toward the top of the display, and 90 TURNT0 points the armadillo toward the right edge of the display, and -90 TURNT0 or 270 TURNT0 both point the armadillo toward the left edge of the display.

TURNTWD (x y --)

Turns the armadillo so that it faces toward the point x,y. "Turn-toward."

DAZM (-- n)

Returns the direction, in degrees (0-359), in which the armadillo is facing. Stands for "dillo azimuth."

DINIT (--)

DINIT stands for "armaDillo INITIALize." Use it to return all options to their default values and to center the armadillo in the display. Useful during practice and experimentation.

DRAW (n --)

Move the armadillo n units in the direction in which it is heading. Draw that portion of the line of travel of the armadillo, including the first point, that falls within the current window, using the current PEN value.

DRAWT0 (x y --)

Move the armadillo to x y and draw that portion of the line of travel that falls within the current window, using the current PEN color register.

PHIL (n --)

Move the armadillo n spaces in the direction it is heading, and as in DRAW, color that portion of the path of travel with the PEN value. Also perform a fill to the right during the time that the armadillo is in the current window.

PHILT0 (x y --)

Move the armadillo to the point x y. Then proceed as in PHIL.

Windows and Coordinate Systems

The following discussion is largely technical. Even so, it may be skimmed by the casual user, who can also get a "hands-on" feel for the operation of windows and coordinate systems by following the examples in the "Strolling Through valGRAPHICS" section of this package. The proliferation of quantities in this package is necessitated by allowing it to handle both "absolute" and "relative" windows at the same time. Because of the complex changes of coordinate system that this entails, a variety of different data are kept on system configuration. This process is transparent to the casual user but may be used with great power by the experienced programmer.

This package uses cartesian (rectangular) coordinate systems (CS's) throughout. For highest speed in graphics work, the graphics coordinate system should be in the same "scale" as the hardware. That is, moving one unit horizontally or vertically in the graphics CS should move the graphics cursor (in this case called the armadillo), one pixel. Doing this avoids additional, usually relatively slow, multiplication and division operations to make the graphics CS "fit" the hardware CS. However, sometimes the speed sacrifice is worthwhile in achieving a desired effect. Therefore, both types of CS are supported in this package. The default CS is of the first type, and it may also be called into play explicitly by the command "MCOOR" which stands for "machine coordinates." This mode is used for high-speed at some sacrifice of flexibility. The optional mode is called by "SCOOR" which stands for "scaled coordinates." Before executing SCOOR, the user may want to set up coordinate boundaries by using SET-SCALE, defined below. Moving between these two types of CS may also be handled automatically by the window routines discussed next. Because of automatic initialization routines in GR., the user may employ both machine and scaled CS's without ever calling them up explicitly. This happens through the commands WINDOW, which puts the system into the machine CS before interpreting its 4 stack arguments; and RELWND, which puts the system into scaled coordinates before interpreting its 4 stack arguments. (Clearly, RELWND must force the system into scaled coordinates, since it will be creating a window with the same numerical coordinates as the one RELWND works from, though the windows will generally be different sizes.) For some help in familiarization with these procedures, please refer to the examples in the "Strolling..." section.

A "window," for the purposes of this package, is a rectangular portion of the graphics display area. Windows are implemented to allow "clipping" as well as some additional scaling and distortion features. Clipping allows the armadillo to travel inside and outside the currently active window, while allowing drawing and filling only while the armadillo is within the boundaries of the window.

The current window's "physical" boundaries are kept in the system quans WNDLFT, WNDRGT, WNDTOP, and WNDBOT. (For an explanation of the QUAN structure, see the section on this topic.) The user does not generally access these quantities directly, but sometimes may want to do so for special effects. WNDLFT and WNDRGT are, respectively, the number of pixels from the left edge of the display to the left edge of the window, and the number of pixels from the left edge of the display to the right edge of the window. Similarly, WNDTOP and WNDBOT are referenced from the top of the display. Again, the user doesn't have to use these quantities; they are, however, the "bedrock" of the windowing process. These quans are in the DILLO vocabulary.

When a graphics-type GR. command is executed (3-12, see GR. above) a window, called the "base window," is set up which takes up the entire graphics display area. (The "physical" edges of the base window are stored in the system quans WNDL, WNDR, WNDT, and WNDB, which have meanings similar to WNDLFT, etc., above, and are likewise in the DILLO vocabulary.) The user may generally return to the base window at any time before leaving the graphics mode by executing BASWND. When the base window is made current by the user explicitly or by GR., the armadillo is placed at the point 0,0, i.e., the center of the window, and turned to 0 degrees, or straight up. The default "numerical" values of the window-boundaries are set so that they correspond to pixel counts vertically and horizontally. For instance, in 7 GR. the numerical boundaries would be +-79 horizontally (since there are 160 pixels across the display in that mode), and +-39 vertically (since the mode is 80 pixels high.) These values are stored in the system quans WNDW, WNDE, WNDN, and WNDS, which stand for "window-west," etc. These values may be altered by means described below (SET-SCALE), although the change will slow down the draw routines because of the extra transformation required when not working in the "natural" coordinates of the system.

After initializing to a graphics mode with GR., the user may use the various commands in this package to create graphics displays in the base window. However, additional flexibility is available to the user by defining new windows, as follows.

The command WINDOW is used to define temporarily a rectangular area of the display as the current window. This definition will last until the next window defining command e.g., WINDOW, BASWND, GR., DINIT, etc. WINDOW defines the window in the coordinate system of the base window. Indeed, WINDOW does BASWND before proceeding. (The base window is set up automatically by GR., or by DEFBAS when using a customized display list.) WINDOW expects four arguments on the stack, namely the left, right, top and bottom edges of the new window, expressed in the coordinate system of the base window. (RELWND ("rel-wind"), defines a window relative to the current window, not the base window; its description otherwise parallels that of WINDOW.) When WINDOW is executed, a new window is made current, and all applicable internal quans are altered as appropriate. The armadillo is centered in the new window and turned to 0 degrees. The numerical boundaries of the new window will be, as stated before, WNDW, WNDE, WNDN, and WNDS.

(Advanced users: NOTE that, when in a GR. mode, decimal 88 @ will leave the address of the byte in the upper-left-hand corner of the display. Internal calculations are based on this location. In general, if the user wishes to redirect the graphics routines in this package to a display memory area in a non-GR. display mode, he or she need do two things: Store the appropriate value into memory location decimal 88, and then execute DEFBAS, described below, to establish a base window. Note, however, that if your display memory makes a discontinuous jump, as can occur for instance when crossing a 4K boundary, the graphics routines will not function properly.)

Additionally, the window-naming word, THISWND, is provided for ease simplicity in returning to a specific window.

Reference on clipping algorithms:

A Practical Introduction to Computer Programs, Ian O. Angell.

va1GRAPHICS GLOSSARY
Part II of III

Windows and Coordinate Systems

WINDOW (left right top bottom --)

Sets a new window whose boundaries, expressed in the coordinate system of the base window (not the current window), are taken from the stack in the order indicated. The armadillo is centered in the new window and turned to a zero angle. Machine coordinates are activated. (See MCOOR).

RELWND (left right top bottom --)

Makes current a window whose edges are as indicated on stack in the coordinate system of the current window (not the base window). Scaled coordinates are activated. (See SCOOR).

WIPE (--)

Colors the entire current window according to the color register selected by the last PHBAK command. Note that since WIPE uses the system routine DRAWLN it will be affected by DRXOR. Hence if ON DRXOR has been executed last then WIPE will XOR all pixels in the entire current window with the value set by PHBAK, rather than replacing them with that value. This is useful for interesting and often eerie effects.

FRAME (--)

Draws a line around the current window according to the color register selected by the last PEN command.

BASWND (--)

Makes the base window (usually the full window first put up by a GR. command) current, centers the armadillo and turns it to 0 degrees.

THISWND xxx, (--)
xxx: (--)

Creates a word, xxx, which when executed makes current the window which was current at the time xxx was defined. Also centers the armadillo and turns it to 0 degrees, and restores XFORM to its state at the time xxx was defined. Located in the "Window Naming" package.

DEFBAS (left right top bottom --)

Advanced users. Used to set up a base window when not using GR.. The values indicated are the number of pixels from the left edge of the display (for left and right) and from the top edge of the display (for top and bottom). Before using this command, the value at decimal 88 should be set to point to the byte that represents the upper-left-hand corner of the display area to be used for graphics. DEFBAS is in the DILLO vocabulary.

SET-SCALE (horiz vert --)

Used to redefine the horizontal and vertical numerical boundaries of windows. After executing SET-SCALE, the SCOOR (stands for "scaled coordinates") command will set windows to range horizontally between +-horiz and vertically between +-vert. Note that the point 0,0 will remain the center point of windows. Since the command RELWND does SCOOR, relative windows will reflect use of SET-SCALE. SET-SCALE is in the DILLO vocabulary.

MCOOR (--)

Sets the horizontal and vertical numerical boundaries of windows to correspond to the number of pixels in each direction in the base window. "MCOOR" stands for "machine coordinates." It is not generally accessed directly by the user, with one exception: After having done a RELWND and returning to the base window by BASWND, an increase in speed may be had by executing MCOOR, if the user was using the default scale set automatically by GR. This is a fine point, but worth noting. MCOOR is in the DILLO vocabulary.

SCOOR (--)

Sets the horizontal and vertical numerical boundaries of windows to correspond to the default values set by GR. or by values set by SET-SCALE. "SCOOR" stands for "scaled coordinates." It is not generally accessed directly by the user. SCOOR is in the DILLO vocabulary.

:WCTR (--)

Centers the armadillo in the current window.

:WCTRO (--)

Centers the armadillo in the current window and turns it to 0 degrees.

Line-naming and line manipulation; point-naming

These packages support labeling various graphics "entities" for convenience in recalling them subsequently, for a variety of purposes.

Lines are stored internally as three-number quantities which are the (non-unique) A, B, and C parameters in standard algebraic line notation. (See the section on The Straight Line in Mathematical Handbook for Scientists and Engineers, 2nd Edition, by Korn and Korn. Point/slope representation is insufficient; point/azimuth representation would work but was not used because of some doubts concerning execution speed.) Labeling of lines is done principally for subsequent geometric-construction-type operations, like finding the intersection of two lines, or the point where the armadillo would intersect a given line.

```
NAMEPT  xxx, ( x y -- )
        xxx: ( -- x y )
```

Creates a word xxx. When xxx is executed, it returns x and y to the stack.

```
THISPT  xxx, ( ---- )
        xxx: ( -- x y )
```

Creates a word xxx. When xxx is executed, it returns to the stack the x and y coordinates of the armadillo in the coordinate system of the window current at the time xxx was created.

```
2PT-LN ( x1 y1 x2 y2 -- a b c )
```

Takes the coordinates of two points on the stack and leaves A, B, and C coefficients of the line connecting two points. "Two-point-line."

```
MAKLN   ( -- a b c )
```

Pushes to stack the A, B, C representation of the imaginary line on which the armadillo is sitting and along which it faces. Useful in finding where the armadillo would intersect a line along its current path. ("Make-line.")

```
NAMELN  xxx, ( a b c -- )
        xxx: ( -- a b c )
```

Creates the word xxx. When xxx is executed, it returns the values a b c to the stack.

```
THISLN  xxx, ( -- )
        xxx: ( -- a b c )
```

Creates the word xxx. When xxx is executed, it returns the A B and C values of the line that the armadillo was sitting on and facing along when xxx was created. ("This-line.")

```
2LNX   ( a1 b1 c1 a2 b2 c2 -- x y )
```

Given two lines on the stack in a b c form, 2LNX returns the point of intersection of the two lines. If the lines are parallel or if their point of intersection is very distant and would cause coordinate overflow, 2LNX will leave -1, -1. ("Two-line-intersection" or "Two-line-X.")

va|GRAPHICS GLOSSARY
Part III of III

Options

The basic commands, followed by the commands that operate the "switches" on options, are described below.

DRAW (n --)

Standard option: Move the armadillo n units in the direction in which it is heading. Draw that portion of the line of travel of the armadillo, including the first point, that falls within the current window, using the current PEN value.

ON DRXOR: XOR pixels with the PEN color instead of overwriting them with the PEN color.

ON DRUNT: Stop on hitting a pixel of the value selected with the last DRBAK or PHBAK command, whichever was last.

OFF DRUNOT: DRUNOT makes a difference only when ON DRUNT has been executed. When DRUNOT is off and DRUNT is on, lines halt upon hitting a pixel of the last color set by DRBAK or PHBAK, whichever was executed last. When DRUNOT is on, which is the default case, and DRUNT is on also, lines will halt upon hitting a pixel not of the last color set by DRBAK or PHBAK, whichever was executed last.

OFF DRIST: Don't draw the first point in a line. Useful when drawing connected lines after ON DRUNT so that the last point of a line won't be interpreted as the stop condition of the next line. See "Strolling..." for an example.

DRAWTO (x y --)

Standard option: Move the armadillo to x y and draw that portion of the line of travel that falls within the current window, using the current PEN color register.

ON DRXOR: XOR pixels with the PEN color instead of overwriting them with the PEN value.

ON DRUNT: Stop on encountering a pixel of the color selected with the last DRBAK or PHBAK command, whichever was last.

OFF DRUNOT: DRUNOT makes a difference only when ON DRUNT has been executed. When DRUNOT is off and DRUNT is on, lines halt upon hitting a pixel of the last color set by DRBAK or PHBAK, whichever was executed last. When DRUNOT is on, which is the default case, and DRUNT is on also, lines will halt upon hitting a pixel not of the last color set by DRBAK or PHBAK, whichever was executed last.

DRAWTO (cont'd)

OFF DR1ST: Don't draw the first point in a line. Useful when drawing connected lines after ON DRUNT so that the last point of a line won't be interpreted as the stop condition of the next line. See "Strolling..." for an example.

?DRSTP (-- f)

?DRSTP is a quan whose value is adjusted after each DRAW and DRAWTO. If ?DRSTP is true (non-zero) then the last DRAW or DRAWTO was terminated because ON DRUNT had been executed and the line-drawing routine encountered a pixel whose value was that selected by the last DRBAK or PHBAK command, whichever was last. ?DRSTP is useful in conjunction with RELOC.

RELOC (--)

Relocates the armadillo to the location of the last pixel drawn by the last DRAW or DRAWTO command. If no points were drawn by the last DRAW or DRAWTO command, (e.g., if the line fell entirely outside the current window) then the armadillo is not moved. RELOC is useful in conjunction with ON DRUNT. See example in "Strolling..." RELOC is in the DILLO vocabulary.

DRAWLN (column row --)

A system routine, not intended for general use. This high-speed routine replaces the DRAWTO routine in valFORTH 1.1, which used the same OS routine as the BASIC DRAWTO command. DRAWLN is in the DILLO vocabulary.

PHIL (n --)

Standard option: Move the armadillo n spaces in the direction it is heading, and ...

As in DRAW, color that portion of the path of travel with the PEN value. Also perform a fill to the right during the time that the armadillo is in the current window. The color of the fill is set either by the PEN value or the PHPEN value, whichever was declared last. The fill will always terminate on reaching the edge of the current window if it has not been terminated prior to this event. The fill will also terminate on reaching a pixel that is not background color. In the standard option, the command ON PHUNT ("phil until") has been executed so that the fill will stop on the pixel of color register set by PHBAK, and 0 PHBAK has been executed so that the actual background register, 0, will also be used as the phil "background" register. ON RPHIL and OFF LPHIL have been executed so that the fill will be toward the right only. ON PH+DR has also been executed so that the line of travel of the armadillo is drawn in addition to the fill operation.

PHIL (cont'd)

Example Options:

1 PHBAK: The fill will now stop on reaching a pixel of color register 1 (in this example), or the edge of the window.

ON PHUNOT 2 PHBAK: The state of PHUNOT only matters if ON PHUNT has been executed. The effect of PHUNOT ("fill until not") is that the fill will now stop on reaching a pixel NOT of color register 2 (in this example), or on reaching the edge of the window.

OFF PHUNT: Turning off fill-until means that now the fill will ONLY stop on reaching the edge of the window.

OFF PH+DR: Turning off PH+DR means that now the routines will not draw the line the armadillo is moving along, and will just fill as indicated.

ON LPHIL: Now the routines will also fill to the left.

OFF RPHIL: Now the routines will not fill to the right.

ON PHXOR: Now the routines will XOR the pixels with the PEN or PHPEN value, whichever was last declared, rather than replacing them with it.

PHILTO (x y --)

Move the armadillo to the point x y. Then proceed as in PHIL.

Options:

(All words below take a flag stack argument, and leave none.)

<u>Switch</u>	<u>Default</u>	<u>ON</u>	<u>OFF</u>
RPHIL	on	Enables right fill with PHIL, PHILTO	Disables right fill with PHIL, PHILTO.
LPFIL	off	Enables left fill with PHIL, PHILTO.	Disables left fill with PHIL, PHILTO.
DRXOR	off	DRAW, DRAWTO will xor pixels with line color.	DRAW, DRAWTO will replace pxls with line color.
PHXOR	off	PHIL, PHILTO will xor pixels with fill color.	PHIL, PHILTO will replace pxls with fill color.
DRUNT	off	Enable draw-until functions.	Disable draw-until functions.
PHUNT	off	Fill to edge of window or to dest. pixel.	Fill until encountering halt pixel cond set by PHBAK, PHUNOT.
DRUNOT	on	With DRUNT on, DRAW, DRAWTO draw until hit color set by DRBAK, PHBAK.	With DRUNT on, DRAW, DRAWTO draw until hit not color set by DRBAK, PHBAK.
PHUNOT	on	With PHUNT on, PHIL, PHILTO fill until hitting color set by PHBAK.	With PHUNT on, PHIL, PHILTO fill until hitting not color set by PHBAK.
PH+DR	on	PHIL, PHILTO draw line as filling.	PHIL, PHILTO don't draw line as filling.
DR1ST	on	First point of lines is drawn.	First point of lines is not drawn.
PHCRNR	off	PHIL, PHILTO perform corner checking, armadillo must be moving vertically.	No corner checking.

DINIT sets all switches to their default values.

Screen Dump

This graphics 8 screen-to-Epson/Grafrax dump routine was contributed by William Volk, who also collaborated on other parts of this package.

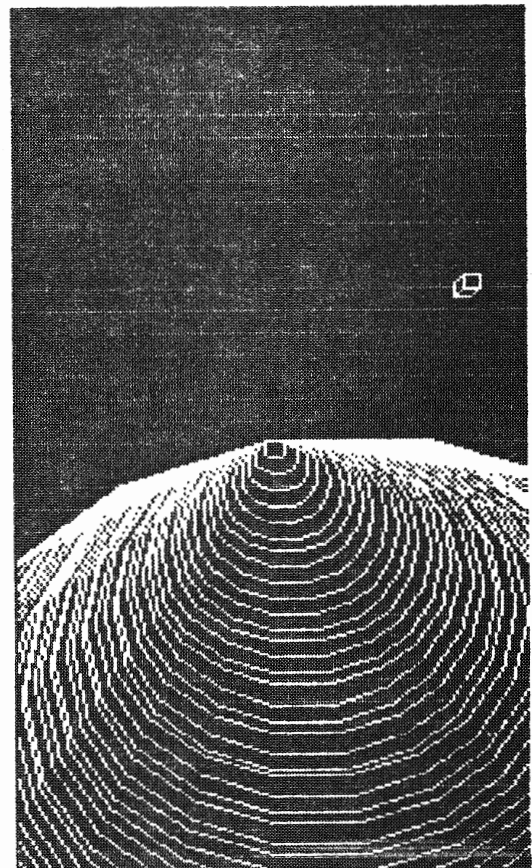
To dump graphics 8 screens (split or full), load this code and execute with GRDUMP. Some samples are shown below.

<pre> Scr # 0 (Dillo: GRDUMP) 1 2 : DMPCOL DILLO (col --) 3 -1 WNDB DO 4 DUP 88 @ + 5 I BYT/LN * + C@ 6 EMIT 7 -1 +LOOP 8 DROP ; 9 10 : GRFLT DILLO (--) 11 ." " 12 27 EMIT 75 EMIT WNDB 13 WNDT - 1+ EMIT 0 EMIT ; 14 15 ==> </pre>	<pre> Scr # 0 (Dillo: BOX-KITE 1 2 : BOX-KITE 3 8 GR. 4 50 0 5 DO 6 I TCIRCLE 7 LOOP 8 -60 60 GOTO 5 QUBE 9 ON DRXOR 1 PHBAK WIPE 10 DINIT ; 11 12 13 14 15 </pre>
---	---

```

Scr #
0 ( Dillo: GRDUMP )
1
2 : GRDUMP DILLO
3 ( turn off screen, on printer )
4 PFLAG @ 2 PFLAG !
5 ( set line/inch = 9 on Epson )
6 27 EMIT 65 EMIT 8 EMIT
7 ( dump the screen )
8 CR BYT/LN 0
9 DO
10 GRFLT I DMPCOL CR
11 LOOP
12 27 EMIT 65 EMIT 12 EMIT
13 CR CR PFLAG ! ;
14
15

```



Interfacing to Custom Display Lists

The advanced user wishing to interface valGraphics to a custom (non-GR.) display list should recognize that any area of display memory in which valGraphics will be required to draw must be continuous. Thus, for example, if a 4K memory boundary is crossed, necessitating a jump instruction in the display list, the user must ensure that display memory itself crosses the 4K boundary smoothly.

The location 88 decimal was used by the Atari OS to point to the byte in display memory corresponding to the upper left corner of the display, and has been adopted for the same purpose in this package. The first thing to do, then, is point 88 to the address in display memory that valGraphics should treat as the upper left corner of its drawing area.

The second step is to set up a base window, much as the GR. provided in this package does. Use the word DEFBAS to do this, as described in the glossary. Note that this word expects its arguments as numbers of pixels, and that "left" and "top" will usually be 0.

Finally, you need to tell the system what graphics mode you're drawing in. The word UGR. (for "user GR.") is provided for this purpose. Give it a number from 3 through 12, and it will set up quans like PX/BYT and so on. UGR. recognizes if you have set up for wide or narrow screen widths, also, and acts accordingly.

Do BASWND and the armadillo is centered, pointed up, and ready.

A note on QUAN structures

The "quan" is a new FORTH data structure, developed at Valpar, and being introduced in this package. Quans were devised to cut down on wasted memory and runtime encountered when using the "variable" data structure. Quans work as follows: (Advanced users may want to follow along in the source code for these structures also.)

Defining a quan:

```
QUAN BINGO
```

Note that quans do not take initial values. This form was chosen to allow for simpler upgrading to target-compiled code later on.

Giving a quan a value:

```
1234 TO BINGO
```

Note that since TO is immediate, "TO BINGO" compiles to only 2 bytes instead of the 4 bytes that would be required if BINGO were a variable (i.e., BINGO !).

Getting a value back from a quan:

```
BINGO
```

Simply saying the name of the quan will leave its value on the stack, in this case 1234. In this way, quans act like constants. BINGO above also compiles to only 2 bytes instead of the 4 required to fetch if it were a variable (i.e., BINGO @).

Getting the address of the data in the quan:

```
AT BINGO
```

This will leave the address of the first byte of data in BINGO on the stack, or compile the address as a literal if encountered during compilation. (AT is immediate.) This is useful for a variety of purposes in general programming and in interfacing to machine language routines.

Advanced users:

The FORTH 83 Standard appears to lean toward "non-state-smart" words, which is proper for target-compiled applications. We expect to support both "state-smart" and "non-state-smart" versions of various words, as appropriate for different users.

Note that while

```
15 AT BINGO +! and 15 BINGO + TO BINGO
```

accomplish the same task and take the same amount of memory, the first version is faster by one primitive nest.

The most significant internal feature of quan is that it has 3 cfa's instead of just the one common to most FORTH words. This initial 4 byte disadvantage is overcome at the second use of a quan, and so poses essentially no problem. CQUAN, 2QUAN, 3QUAN, etc., have also been implemented, and the user may have some fun puzzling these out before they are published elsewhere. Note that a 2quan takes 2 arguments from the stack when used with T0, and leaves 2 when used alone. When used with AT, a 2quan still leaves the address of the first byte of its "parameter field," as does QUAN. Also, when defining CQUAN it is probably a good idea to still allot 2 bytes for data, so that +! can be used without fear of negative stack arguments. Another new defining structure is called "FLAG." Flags have only two cfa's, dropping the one that supports the "AT" function. Flags keep only one byte of data, a flag; hence they are 3 bytes shorter than quans. Flags would not be used in this package enough to justify the additional code, but may be worthwhile in other applications.

Higher speed and cleaner array structures may also be implemented using the quan strategy, and may be included in a future release of our utilities-editor package. (This would be made available to current u/e owners at a price-difference-plus-handling charge.)

The word VECT has also been introduced in this package. It has two cfa's, and replaces the rather cumbersome variable-based vectoring procedure,

```
' SOMEWORD CFA SOMEVARIABLE ! and  
SOMEVARIABLE @ EXECUTE
```

with the cleaner, faster, and memory-shorter

```
' SOMEWORD CFA TO SOMEVECT and  
SOMEVECT
```

(intentionally left blank)

QUICK TRIG

Since floating point trigonometric operations on the Atari machines are rather slow and provide accuracy unnecessary for many applications, this package provides integer versions of sine, cosine, and arctangent functions that run much faster than their floating point cousins.

QSIN and QCOS expect scaled radian arguments in the range ± 31416 , ($\pm\pi$), with 10000 representing one radian.

Similarly, QATN returns scaled radian arguments in range ± 15708 ($\pm\pi/2$). QATN accepts arguments in the full single number range, again interpreting 10000 as 1. This at first glance seems to be a significant limitation on QATN's input range but is circumvented by the existence of the more useful QATN2. QATN2 is a four-quadrant arctangent function. It accepts two stack arguments, which may be thought of as "delta-x" and "delta-y," and uses these arguments to construct a value to be used by QATN. QATN2 then performs sign corrections as necessary and returns a value in the range ± 31416 . QATN2 is what is actually used in graphics work, and is used in the word TURNTWD ("turn toward") elsewhere in this package.

For user convenience, the words \rightarrow QRD and \rightarrow QDG are used to convert from scaled-degree arguments to scaled radian arguments and back again.

16K/ (d -- n)

This is a special-purpose high-speed routine that may find other uses. It divides a double number by 16384 and leaves a single number result. Used to speed quick-trig functions.

QSIN (scaled-radians -- scaled-sine)

Takes a scaled-radian argument (range ± 31416) and leaves the scaled sine in the range ± 10000 .

QCOS (scaled-radians -- scaled-cosine)

Takes a scaled-radian argument (range ± 31416) and leaves the scaled cosine in the range ± 10000 .

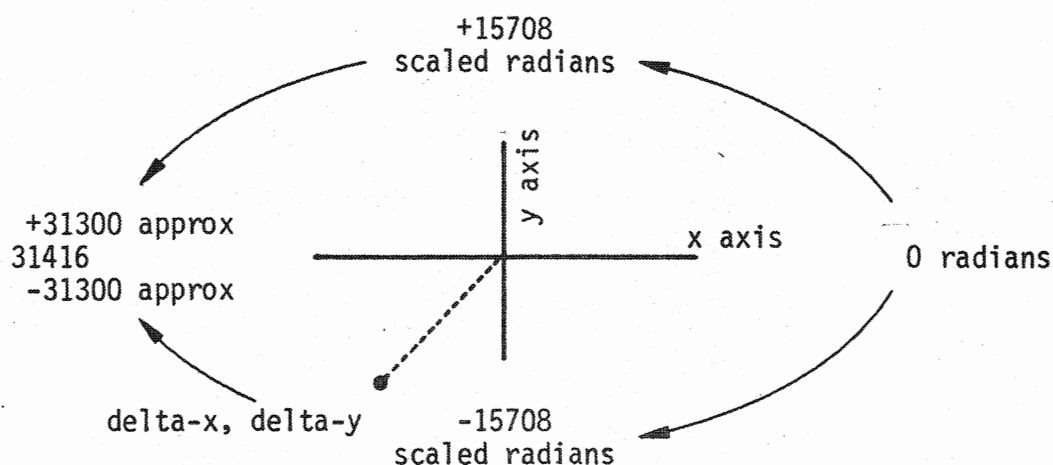
QATN (scaled-argument -- scaled-radians)

Takes a scaled-argument (range 0 to 10000) and leaves the scaled arctangent in the range ± 15708 ($\pm\pi/2$, scaled).

QATN2 ("delta-x" "delta-y" -- scaled-radians)

Assuming that the "x" axis points toward zero radians (zero degrees) on one end and π radians (180 degrees) on the other, QATN2 leaves the angle (range ± 31416) between a line from the origin to the point (delta-x, delta-y).

Counterclockwise angles are positive.



In the illustration above, $\arctan2$ of (delta-x, delta-y) would be approximately -2.1 radians, or -21000 as computed by QATN2.

->QRD (scaled-degrees -- scaled-radians)

Takes a scaled-degree argument (range +-18000) and converts it to a scaled-radian argument (range +- 31416).

->QDG (scaled radians -- scaled-degrees)

Takes a scaled-radian argument (range +-31416) and converts it to a scaled-degree argument. (range +-18000).

FLOATING POINT EXTENSIONS, INCLUDING TRIG FUNCTIONS

This section contains extensions to the Floating Point package provided with valFORTH 1.1. Source Code is on the valGraphics disk. The pages have been numbered so that it may be separated from the rest of the Turtle Graphics package and integrated into the original Floating Point package for ease of reference. Note that the trigonometric functions expect their arguments in radians, as is normal in floating point. Conversion words DG→RD and RD→DG have been provided for convenience, however.

GLOSSARY

PI, PI/2, PI/3, PI/4, and PI/6 (-- fp)

leave their normal floating point values on the stack.

EXP1 (-- fp)

leaves the value of the constant "e" on the stack. Though this name is slightly obscure, it was selected to avoid name collisions.

FP0 and FP1 (-- fp)

have been defined as FP 0 and FP 1, respectively, to reduce memory requirements.

RD/DG and DG/RD (-- fp)

are constants equal to the number of radians in a degree and the number of degrees in a radian, respectively.

FTOP and FBOT (-- fp)

are, approximately, the largest and smallest non-zero absolute values that the machine can represent.

FMINUS (fp1 -- fp2)

Leaves the negative of fp1 as fp2.

2FDUP (fp1 fp2 -- fp1 fp2 fp1 fp2)

Duplicates the two fp values on the top of stack.

F+! (fp1 addr --)

Add fp1 to the fp value at address addr, and leaves result at addr.

FMAX (fp1 fp2 -- fp3)

Leaves the maximum of fp1 and fp2 as fp3.

FMIN (fp1 fp2 -- fp3)

Leaves the minimum of fp1 and fp2 as fp3.

F0< (fp1 -- flag)

Leaves a true flag if fp1 is negative; otherwise, leaves a false flag.

FABS (fp1 -- fp2)

Leaves the absolute value of fp1 as fp2.

2FDROP (fp1 fp2 --)

Discards the two floating point numbers on top of stack.

F, (fp1 --)

Compiles the floating point number on top of stack into the dictionary.

F>R (fp1 --)

Sends fp1 to the return stack.

FR> (-- fp1)

Retrieves the top 6 bytes of the return stack as fp1.

FR (-- fp1)

Copies the top 6 bytes of the return stack as fp1.

F.S (--)

Does a non-destructive printout of the stack assuming all fp numbers.

FPICK (fpn...fp1 n -- fpn..fp1 fpn)

Copies the nth fp number to top of stack. Note that 1 FPICK is the same as FDUP and 2 FPICK is the same as FOVER.

FROLL (fpn fpn-1..fp1 n -- fpn-1..fp1 fpn)

Pulls the nth fp number out of the stack and moves it to top of stack. Note that 2 FROLL is the same as FSWAP and 3 FROLL would be the same as FROT.

-FIX (fp1 -- n)

Converts a fp number to a single number. Accepts positive or negative fp. For large fp numbers, represents to most positive or most negative n, as appropriate.

-FLOAT (n -- fp1)

Converts a single number to a fp number. Accepts positive or negative values for n.

F*OV (fp1 fp2 -- fp3)

Multiplies fp1 and fp2 to produce fp3. If fp3 would overflow or underflow, leaves +- greatest fp number or fp 0 as required.

F/OV (fp1 fp2 -- fp3)

Divides fp1 by fp2 to produce fp3. If fp3 would overflow or underflow, leaves +- greatest fp number or fp 0 as required.

1/FP (fp1 -- fp2)

Leaves the inverse of fp1 as fp2, using F/OV.

->RD (degrees -- radians)

Converts the argument "degrees" into radians. "To radians."

->DG (radians -- degrees)

Converts the argument "radians" into degrees. "To degrees."

SIN (fp1 -- fp2)

Leaves the sine of fp1 as fp2.

COS (fp1 -- fp2)

Leaves the cosine of fp1 as fp2.

TAN (fp1 -- fp2)

Leaves the tangent of fp1 as fp2.

(NOTE: SIN, COS, and TAN expansions fail for fp1 above approximately 2E5 (200,000) radians or 1E7 (10,000,000) degrees.)

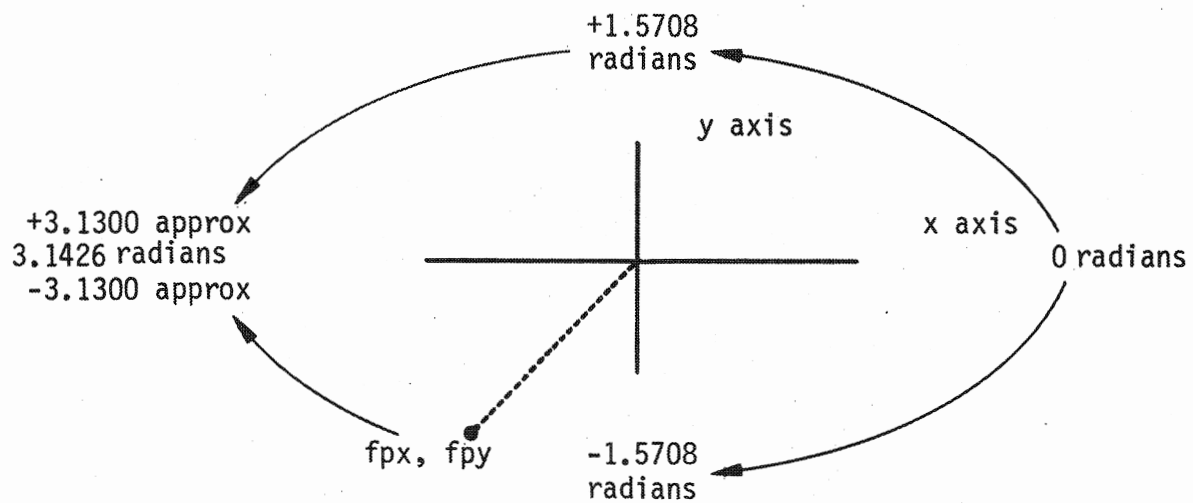
ATN (fp1 -- fp2)

Leaves the arctangent of fp1 as fp2.

ATN2 (fpx fpy -- fp1)

Leaves the four-quadrant arctangent of fpx and fpy as fp1. Assuming that the "x" axis points toward zero radians (zero degrees) on one end and pi radians (180 degrees) on the other, ATN2 leaves the angle, in radians, between a line from the origin to the point (fpx, fpy).

Counterclockwise angles are positive.



In the illustration above, $ATN2$ of (fpx, fpy) would be approximately -2.1 radians.

Reference:

Software Manual for the Elementary Functions by William J. Cody, Jr.
and William Waite.

XLIV va1GRAPHICS SUPPLIED SOURCE LISTING

Screen: 1

```

0 ( Dillo: Drawline routine )
1
2 HEX DILLO DEFINITIONS
3
4 LABEL INCMOD ( sys )
5 E6 C, C4 C, E6 C, C3 C, D0 C,
6 02 C, E6 C, C2 C, A5 C, C2 C,
7 F0 C, 02 C, 24 C, C4 C, 30 C,
8 22 C, 85 C, C6 C, A5 C, C3 C,
9 85 C, C7 C, 84 C, C5 C, A9 C,
10 10 C, 85 C, C1 C, 06 C, C7 C,
11 26 C, C6 C, 26 C, C5 C, 38 C,
12 A5 C, C5 C, E5 C, C4 C, 90 C,
13 04 C, 85 C, C5 C, E6 C, C7 C,
14 C6 C, C1 C, D0 C, EB C, 60 C,
15 ==>

```

Screen: 4

```

0 ( Dillo: Drawline routine )
1
2 LABEL PHTST ( sys )
3 E6 C, C1 C, B1 C, C8 C, 3D C,
4 MTBL2 , 8D C, DRWRK 4 + , AD
5 C, AT PHUDAT , 3D C, MTBL2 ,
6 CD C, DRWRK 4 + , D0 C, 07 C,
7 2C C, DRWSTT 6 + , 30 C, 09 C,
8 10 C, 05 C, 2C C, DRWSTT 6 + ,
9 10 C, 02 C, C6 C, C1 C, 60 C,
10
11
12
13
14
15 -->

```

Screen: 2

```

0 ( Dillo: Drawline routine )
1
2 84 C, C6 C, 84 C, C7 C, A5 C,
3 C3 C, 85 C, C5 C, E6 C, C7 C,
4 38 C, A5 C, C5 C, E5 C, C4 C,
5 85 C, C5 C, C5 C, C4 C, B0 C,
6 F3 C, 60 C,
7
8
9 LABEL BUMPY ( sys )
10 18 C, AD C, DRWRK , 65 C,
11 C8 C, 85 C, C8 C, AD C, DRWRK
12 1+ , 65 C, C9 C, 85 C, C9 C,
13 E6 C, 5A C, 2C C, DRWRK 1+ ,
14 10 C, 04 C, C6 C, 5A C, C6 C,
15 5A C, 60 C, ==>

```

Screen: 5

```

0 ( Dillo: Drawline routine )
1
2 LABEL (DOPHL) ( sys )
3 A5 C, 5B C, 48 C, A5 C, 5C C,
4 48 C, A5 C, C8 C, 48 C, A5 C,
5 C9 C, 48 C, 8A C, 48 C, 84 C,
6 C1 C, 20 C, PHTST , 20 C,
7 BUMPX , 2C C, DRWSTT 4 + ,
8 30 C, 36 C, 2C C, DRWSTT 7 + ,
9 10 C, 31 C, A5 C, C8 C, 48 C,
10 A5 C, C9 C, 48 C, 2C C, DRWRK
11 3 + , 10 C, 08 C, 20 C,
12 BUMPY , EE C, DRWRK 3 + ,
13 F0 C, 0F C, 38 C, A5 C, C8 C,
14 ED C, DRWRK , 85 C, C8 C,
15 ==>

```

Screen: 3

```

0 ( Dillo: Drawline routine )
1
2 LABEL BUMPX ( sys )
3 2C C, DRWRK 2+ , 30 C, 16 C,
4 E8 C, 8A C, CD C, AT PX/BYT ,
5 90 C, 08 C, A2 C, 00 C, E6 C,
6 C8 C, D0 C, 02 C, E6 C, C9 C,
7 E6 C, 5B C, D0 C, 02 C, E6 C,
8 5C C, 60 C, CA C, 10 C, 0C C,
9 AE C, AT PX/BYT , CA C, A5 C,
10 C8 C, D0 C, 02 C, C6 C, C9 C,
11 C6 C, C8 C, A5 C, 5B C, D0 C,
12 02 C, C6 C, 5C C, C6 C, 5B C,
13 60 C,
14
15 ==>

```

Screen: 6

```

0 ( Dillo: Drawline routine )
1
2 A5 C, C9 C, ED C, DRWRK 1+ ,
3 85 C, C9 C, 20 C, PHTST ,
4 68 C, 85 C, C9 C, 68 C, 85 C,
5 C8 C, A5 C, C1 C, C9 C, 02 C,
6 F0 C, 46 C, 38 C, A5 C, 5B C,
7 ED C, AT WNDLFT , A5 C, 5C C,
8 ED C, AT WNDLFT 1+ , 30 C,
9 39 C, 38 C, AD C, AT WNDRGT ,
10 E5 C, 5B C, AD C, AT WNDRGT 1+
11 , E5 C, 5C C, 30 C, 2C C,
12 2C C, DRWSTT 4 + , 30 C,
13 09 C, 84 C, C1 C, 20 C, PHTST
14 , A5 C, C1 C, D0 C, 1E C,
15 -->

```

Screen: 7

```

0 ( Dillo: Drawline routine )
1
2 B1 C, C8 C, 2C C, DRWSTT 5 + ,
3 30 C, 03 C, 3D C, MTBL1 ,
4 85 C, C1 C, AD C, AT PHDAT ,
5 3D C, MTBL2 , 45 C, C1 C,
6 91 C, C8 C, 84 C, C1 C, 20 C,
7 BUMPX , 4C C, HERE 44 - ,
8 68 C, AA C, 68 C, 85 C, C9 C,
9 68 C, 85 C, C8 C, 68 C, 85 C,
10 5C C, 68 C, 85 C, 5B C, 60 C,
11
12
13
14
15 ==>

```

Screen: 8

```

0 ( Dillo: Drawline routine )
1
2 LABEL (PHIL) ( sys )
3 2C C, DRWRK 5 + , 10 C, 20 C,
4 AD C, DRWRK 2+ , 48 C, 2C C,
5 DRWSTT 1+ , 10 C, 06 C, 8C C,
6 DRWRK 2+ , 20 C, (DOPHL) ,
7 2C C, DRWSTT 2+ , 10 C, 08 C,
8 A9 C, FF C, 8D C, DRWRK 2+ ,
9 20 C, (DOPHL) , 68 C, 8D C,
10 DRWRK 2+ , 60 C,
11
12
13
14
15 ==>

```

Screen: 9

```

0 ( Dillo: Drawline routine )
1
2 LABEL PIXEL ( sys )
3 B5 C, 00 C, 85 C, 5A C, 85 C,
4 C4 C, B5 C, 02 C, 85 C, 5B C,
5 B5 C, 03 C, 85 C, 5C C, 84 C,
6 C2 C, 84 C, C3 C, A2 C, 08 C,
7 A5 C, C4 C, 29 C, 01 C, F0 C,
8 0E C, 18 C, AD C, AT BYT/LN ,
9 65 C, C3 C, 85 C, C3 C, A9 C,
10 00 C, 65 C, C2 C, 85 C, C2 C,
11 46 C, C2 C, 66 C, C3 C, 66 C,
12 C4 C, CA C, D0 C, E3 C, 18 C,
13 A5 C, 58 C, 65 C, C4 C, 85 C,
14
15 -->

```

Screen: 10

```

0 ( Dillo: Drawline routine )
1
2 C8 C, A5 C, 59 C, 65 C, C3 C,
3 85 C, C9 C, AD C, AT PX/BYT ,
4 85 C, C4 C, A5 C, 5B C, 85 C,
5 C3 C, A5 C, 5C C, 85 C, C2 C,
6 20 C, INCMOD 8 + , 18 C, A5 C,
7 C7 C, 65 C, C8 C, 85 C, C8 C,
8 A5 C, C9 C, 69 C, 00 C, 85 C,
9 C9 C, A6 C, C5 C, 60 C,
10
11
12
13
14
15 ==>

```

Screen: 11

```

0 ( Dillo: Drawline routine )
1
2 LABEL DTST ( sys )
3 E6 C, C1 C, AD C, AT DRUDAT ,
4 3D C, MTBL2 , 8D C, DRWRK 4 +
5 , B1 C, C8 C, 3D C, MTBL2 ,
6 CD C, DRWRK 4 + , D0 C, 07 C,
7 2C C, DRWSTT B + DUP , 30 C,
8 07 C, 10 C, 06 C, 2C C, ,
9 30 C, 01 C, 60 C, C6 C, C1 C,
10 AD C, DRWRK F + , F0 C, 42 C,
11 EE C, AT ?DRSTP , C8 C, 8C C,
12 DRWRK E + , 88 C, AD C, DRWRK
13 2+ , 48 C, 20 C, BUMPX ,
14 B1 C, C8 C, 3D C, MTBL2 ,
15 -->

```

Screen: 12

```

0 ( Dillo: Drawline routine )
1
2 8D C, DRWRK 4 + DUP , AD C,
3 AT DRUDAT , 3D C, MTBL2 ,
4 CD C, , D0 C, 07 C, 2C C,
5 DRWSTT B + DUP , 30 C, 0D C,
6 10 C, 05 C, 2C C, , 10 C, 06
7 C, CE C, DRWRK E + , CE C,
8 AT ?DRSTP , 68 C, 48 C,
9 49 C, FF C, 8D C,
10 DRWRK 2+ DUP , 20 C, BUMPX ,
11 68 C, 8D C, , 60 C,
12
13
14
15 ==>

```

Screen: 13

```

0 ( Dillo: Drawline routine )
1
2 CODE DRAWLN DILLO ( x y -- )
3 86 C, D1 C, 8C C, DRWRK 1+ ,
4 8C C, DRWRK 2+ , AD C,
5 AT BYT/LN , 8D C, DRWRK DUP ,
6 38 C, B5 C, 02 C, E5 C, 5B C,
7 8D C, DUP 6 + , B5 C, 03 C,
8 E5 C, 5C C, 8D C, DUP 7 + ,
9 10 C, 12 C, CE C, DUP 2+ ,
10 38 C, 98 C, ED C, DUP 6 + ,
11 8D C, DUP 6 + , 98 C, ED C,
12 DUP 7 + , 8D C, DUP 7 + , AD
13 C, DUP 6 + , 0D C, DUP 7 + ,
14 8D C, DUP F + , 38 C, B5 C,
15 -->

```

Screen: 14

```

0 ( Dillo: Drawline routine )
1
2 00 C, E5 C, 5A C, 8D C, DUP
3 8 + , 98 C, E9 C, 00 C, 8D C,
4 DUP 9 + , 10 C, 1A C, 38 C,
5 98 C, ED C, DUP 8 + , 8D C,
6 DUP 8 + , 98 C, ED C, DUP 9 +
7 DUP , 8D C, , CE C, DUP 1+ ,
8 38 C, 98 C, ED C, DUP , 8D C,
9 DUP , A5 C, 5A C, 85 C, C4 C,
10 20 C, PIXEL E + , 8C C, DUP
11 A + , 84 C, C2 C, AD C, DUP
12 8 + , 85 C, C3 C, AD C, DUP
13 6 + , 85 C, C4 C, C5 C, C3 C,
14 B0 C, 05 C, AD C, DUP 7 + ,
15 ==>

```

Screen: 15

```

0 ( Dillo: Drawline routine )
1
2 F0 C, 11 C, CE C, DUP A + ,
3 A5 C, C4 C, 85 C, C3 C, AD C,
4 DUP 7 + , 85 C, C2 C, AD C,
5 DUP 8 + , 85 C, C4 C, 20 C,
6 INCMOD , A5 C, C4 C, A5 C,
7 C4 C, 8D C, DUP B + , 8D C,
8 DUP C + , ) 84 C, C4 C, A9 C,
9 FF C, 8D C, DUP 3 + , 8C C,
10 AT ?DRSTP , 8C C, DUP D + ,
11 2C C, DRWSTT A + , 10 C, 03 C,
12 CE C, DUP D + , A5 C, C6 C,
13 85 C, C2 C, A5 C, C7 C, 85 C,
14 C3 C, 18 C, A5 C, C5 C, 65 C,
15 ( WAS 46 ) -->

```

Screen: 16

```

0 ( Dillo: Drawline routine )
1
2 C4 C, B0 C, 05 C, CD C, DUP
3 C + DUP , 90 C, 09 C, ED C,
4 , E6 C, C3 C, D0 C, 02 C, E6 C,
5 C2 C, 85 C, C4 C, A9 C, FF C,
6 8D C, DUP 5 + , 2C C, DUP D +
7 , 30 C, 36 C, 2C C, DRWSTT 9 + ,
8 10 C, 0E C, 84 C, C1 C, 20 C,
9 DTST , A5 C, C1 C, F0 C, 05 C,
10 EE C, AT ?DRSTP , D0 C, 76 C,
11 2C C, DRWSTT , 10 C, 08 C,
12 20 C, (PHIL) , 2C C, DRWSTT
13 3 + , 30 C, 16 C, B1 C, C8 C,
14 2C C, DRWSTT 8 + , 30 C, 03 C,
15 ==>

```

Screen: 17

```

0 ( Dillo: Drawline routine )
1
2 3D C, MTBL1 , 85 C, C1 C,
3 AD C, AT DRDAT , 3D C, MTBL2 ,
4 45 C, C1 C, 91 C, C8 C, AD C,
5 DUP A + , D0 C, 06 C, 20 C,
6 BUMPY , 4C C, HERE 8 + , 20 C,
7 BUMPX , 8C C, DUP 5 + , AD C,
8 AT ?DRSTP , D0 C, 3D C, A5 C,
9 C3 C, D0 C, 02 C, C6 C, C2 C,
10 C6 C, C3 C, A5 C, C2 C, 05 C,
11 C3 C, D0 C, A6 C, AD C, DUP
12 A + , D0 C, 06 C, 20 C, BUMPX
13 , 4C C, HERE 5 + , 20 C, BUMPY ,
14 8C C, DUP D + , CE C, B + ,
15 -->

```

Screen: 18

```

0 ( Dillo: Drawline routine )
1
2 F0 C, 03 C, 4C C, HERE 9A - ,
3 A9 C, FF C, 4D C, DRWRK 1+ ,
4 8D C, DRWRK 1+ , 20 C, BUMPY ,
5 A9 C, FF C, 4D C, DRWRK 2+ ,
6 8D C, DRWRK 2+ , 20 C, BUMPX ,
7 A6 C, D1 C, 4C C,
8 ASSEMBLER POPTWO ,
9
10 C;
11
12
13 DCX
14
15 ==>

```

Screen: 19

```
0 ( Dillo: PLOT )
1
2 : PLOT DILLO ( xa ya -- )
3 [ DRWSTT 9 + ]L >R
4 R @ 0 R ! >R
5 2DUP 90 C! 91 ! DRAWLN
6 R> R> ! ;
7
8
9
10
11
12
13
14
15 -->
```

Screen: 22

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 20

```
0 ( Dillo: [LOOK] )
1
2 HEX
3 CODE (LOOK) ( xa ya -- px# px1 )
4 86 C, XSAVE C, 20 C, PIXEL ,
5 B1 C, N 6 + C, 3D C, MTBL2 ,
6 48 C, 8A C, A6 C, XSAVE C,
7 95 C, 02 C, A9 C, 00 C,
8 95 C, 03 C, 68 C, 95 C, 00 C,
9 4C C, NEXT , C;
10 DCX
11
12
13
14
15 ==>
```

Screen: 23

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 21

```
0 ( Dillo: LOOK )
1
2 FORTH DEFINITIONS
3
4 : LOOK DILLO ( x y -- pen )
5 91 @ 90 C@ 2SWAP XF/L
6 (LOOK) SWAP 8 PX/BYT / >R R *
7 8 - R> + SHIFT
8 (ROT 90 C! 91 ! ;
9
10
11
12
13
14
15
```

Screen: 24

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 25

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 28

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 26

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 29

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 27

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 30

0 (Qtrig: 16K/)
1 HEX
2 CODE 16K/
3 36 C, 03 C,
4 36 C, 00 C, 36 C, 01 C,
5 36 C, 03 C, 36 C, 00 C,
6 36 C, 01 C, 85 C, 00 C, 95 C,
7 02 C, 85 C, 01 C, 95 C,
8 03 C, 4C C, POP , C;
9
10 DCX
11
12
13
14
15

=>

Screen: 31

```

0 ( Qtrig: [QS/C] )
1
2 : (QS/C) ( rad -- n )
3 DUP 670 )
4 IF DUP 966 )
5 IF 16384 10000 */
6 DUP DUP 4 / M* 16K/
7 11 ( TR4 ) OVER M* 16K/
8 -208 ( TR3 ) + OVER M* 16K/
9 2184 ( TR2 ) + OVER M* 16K/
10 -10923 ( TR1 ) + M* 16K/
11 OVER M* 16K/ +
12 10000 M* 16K/ 1+
13 ELSE 1-
14 ENDIF
15 ENDIF ; -->

```

Screen: 32

```

0 ( Qtrig: QSIN QCOS )
1
2 : QSIN ( rad -- n )
3 DUP 0( >R ABS DUP 15708 )
4 IF 31416 SWAP - ENDIF
5 (QS/C) R)
6 IF MINUS ENDIF ;
7
8 : QCOS ( rad -- n )
9 15708 SWAP ABS DUP 15708 )
10 DUP >R
11 IF SWAP
12 ENDIF - (QS/C) R)
13 IF MINUS ENDIF ;
14
15 ==>

```

Screen: 33

```

0 ( Qtrig: ->QRD ->QDG )
1
2 : ->QRD ( scaled degrees -- )
3 ( scaled radians )
4 31416 18000 */ ;
5
6 : ->QDG ( scaled radians -- )
7 ( scaled degrees )
8 18000 31416 */ ;
9
10
11 ' ( (QATN) ;S ) ( )
12
13
14
15 -->

```

Screen: 34

```

0 ( Qtrig: [QATN] )
1
2 : (QATN) ( n -- rad )
3 0 >R
4 DUP 2679 )
5 IF DUP 17321 10000 */ 10000 -
6 SWAP 17321 + 10000 SWAP */
7 R) 1+ >R
8 ENDIF
9 DUP DUP 10000 */ DUP DUP
10 -509 10000 */ -4708 +
11 SWAP 2/ 7603 +
12 */ 2/ OVER
13 10000 */ +
14 R)
15 IF 5236 + ENDIF ; ==>

```

Screen: 35

```

0 ( Qtrig: QATN )
1
2 : QATN ( n -- n )
3 DUP 0( >R ABS (QATN) R)
4 IF MINUS ENDIF ;
5
6
7
8
9
10
11
12
13
14
15 -->

```

Screen: 36

```

0 ( Qtrig: QATN2 )
1
2 : QATN2 ( x y -- rad )
3 SWAP 2DUP 0( >R 0( >R
4 ABS SWAP ABS SWAP
5 2DUP > DUP >R
6 IF SWAP ENDIF
7 10000 SWAP */ (QATN) R)
8 IF 15708 SWAP -
9 ENDIF I'
10 IF 31416 I 0=
11 IF SWAP ENDIF -
12 ELSE I
13 IF MINUS ENDIF
14 ENDIF R) DROP R) DROP ;
15

```

Screen: 37

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 38

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 39

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 40

0 (Dillo: DILLO [ARMADILLO])
1
2 FORTH DEFINITIONS
3
4 ' (QUAN) (57 KLOAD)
5
6 VOCABULARY DILLO IMMEDIATE
7 (ARMADILLO)
8
9
10
11
12
13
14
15 ==>

Screen: 41

0 (Dillo: quans)
1
2 QUAN DAZM
3 QUAN DX1 QUAN DY1
4
5 DILLO DEFINITIONS
6
7 QUAN WNDLFT QUAN WNRGT
8 QUAN WNDTOP QUAN WNBOT
9 QUAN WNDL QUAN WNDR
10 QUAN WNDT QUAN WNDB
11 QUAN WNDW QUAN WNDE
12 QUAN WNDN QUAN WNDS
13 QUAN DX2 QUAN DY2
14 QUAN ?DOWN QUAN DFLG
15 QUAN ?MCOOR QUAN ?XFM -->

Screen: 42

0 (Dillo : quans etc.)
1
2 QUAN X1D
3 QUAN Y1D
4 QUAN X2D
5 QUAN Y2D
6 QUAN IX1
7 QUAN IY1
8 QUAN IX2
9 QUAN IY2
10 VECT XF/L VECT <XF/L
11 VECT CASP
12 LABEL DRWRK 16 ALLOT
13 LABEL DRWSTT 16 ALLOT
14 DRWSTT 16 ERASE
15 ==>

Screen: 43

```
0 ( Dillo: quans etc. )
1
2 QUAN DRCLR
3 QUAN PHCLR
4 QUAN DRUCLR
5 QUAN PHUCLR
6 QUAN DRDAT
7 QUAN PHDAT
8 QUAN DRUDAT
9 QUAN PHUDAT
10 QUAN BYT/LN
11 QUAN PX/BYT
12 LABEL MTBL1 8 ALLOT
13 LABEL MTBL2 8 ALLOT
14
15 -->
```

Screen: 44

```
0 ( Dillo: quans etc. )
1
2 QUAN ?DRSTP 0 TO ?DRSTP
3 QUAN COSFAC 1000 TO COSFAC
4 QUAN HSCL 1000 TO HSCL
5 QUAN VSCL 1000 TO VSCL
6 QUAN HABS QUAN VABS
7 QUAN HUSR 1000 TO HUSR
8 QUAN VUSR 1000 TO VUSR
9
10
11
12
13
14
15 ==>
```

Screen: 45

```
0 ( Dillo: SHIFT )
1
2 FORTH DEFINITIONS
3
4 HEX
5 CODE SHIFT
6 B4 C, 00 C, 10 C, 0A C, 56 C,
7 03 C, 76 C, 02 C, C8 C, D0 C,
8 F9 C, 4C C, HERE C + , F0 C,
9 08 C, 16 C, 02 C, 36 C, 03 C,
10 88 C, 4C C, HERE 8 - ,
11 4C C, POP , C;
12
13
14
15 -->
```

Screen: 46

```
0 ( Dillo: SWAP- JL )
1
2 CODE SWAP-
3 38 C, B5 C, 00 C, F5 C, 02 C,
4 48 C, B5 C, 01 C, F5 C, 03 C,
5 4C C, BINARY , C;
6
7 DCX
8
9 : JL ( n -- )
10 [COMPILE] ]
11 [COMPILE] LITERAL ; IMMEDIATE
12
13
14 ( DRAWLN : )
15 1 LOAD ==>
```

Screen: 47

```
0 ( Dillo: phil/draw options )
1
2 : RPHIL DILLO ( f -- )
3 MINUS [ DRWSTT 1+ JL C! ;
4
5 : LPHIL DILLO ( f -- )
6 MINUS [ DRWSTT 2+ JL C! ;
7
8 : PH+DR DILLO ( f -- )
9 NOT MINUS [ DRWSTT 3 + JL C! ;
10
11 : PHUNT DILLO ( f -- )
12 NOT MINUS [ DRWSTT 4 + JL C! ;
13
14 : PHXOR DILLO ( f -- )
15 MINUS [ DRWSTT 5 + JL C! ; -->
```

Screen: 48

```
0 ( Dillo: phil/draw options )
1
2 HEX
3
4 : PHUNOT DILLO ( f -- )
5 NOT MINUS [ DRWSTT 6 + JL C! ;
6
7 : PHCRNR DILLO ( f -- )
8 MINUS [ DRWSTT 7 + JL C! ;
9
10 : DRXOR DILLO ( f -- )
11 MINUS [ DRWSTT 8 + JL C! ;
12
13 : DRUNT DILLO ( f -- )
14 MINUS [ DRWSTT 9 + JL C! ;
15 ==>
```

Screen: 49

```

0 ( Dillo: DR1ST DRUNOT )
1
2 : DR1ST DILLO ( f -- )
3 NOT MINUS [ DRWSTT A + ]L C! ;
4
5 : DRUNOT DILLO ( f -- )
6 NOT MINUS [ DRWSTT B + ]L C! ;
7
8
9
10
11
12
13
14 DCX
15 -->

```

Screen: 52

```

0 ( Dillo: ADJCLR MSKTBL )
1
2 : ADJCLR ( -- )
3 AT DRDAT DRCLR (ADJC)
4 AT PHDAT PHCLR (ADJC)
5 AT DRUDAT DRUCLR (ADJC)
6 AT PHUDAT PHUCLR (ADJC) ;
7
8 HEX
9 LABEL MSKTBL
10 7F C, BF C, DF C, EF C,
11 F7 C, FB C, FD C, FE C,
12 3F C, CF C, F3 C, FC C,
13 0F C, F0 C,
14 DCX
15 ==>

```

Screen: 50

```

0 ( Dillo: DTBL )
1
2 DILLO DEFINITIONS
3
4 '( TABLE ==> ) ( )
5 HEX
6 : DTBL
7 CREATE SMUDGE
8 ;CODE 16 C, 00 C, 36 C, 01 C,
9 CA C, CA C, 18 C, A5 C,
10 W C, 69 C, 02 C, 95 C, 00 C,
11 98 C, 65 C, W 1+ C, 95 C,
12 01 C, 4C C, ' + , C;
13 DCX
14
15 ==>

```

Screen: 53

```

0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->

```

Screen: 51

```

0 ( Dillo: [ADJC] )
1
2 : (ADJC) ( adr val -- )
3 8 PX/BYT 8 MIN 0 MAX / SWAP
4 8 3 PICK - SHIFT 255 AND DUP
5 8 4 PICK
6 DO
7 3 PICK MINUS SHIFT
8 SWAP 0+S
9 3 PICK
10 +LOOP
11 DROP SWAP DROP SWAP C! ;
12
13
14
15 -->

```

Screen: 54

```

0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 ==>

```

Screen: 55

```
0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->
```

Screen: 58

```
0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 ==>
```

Screen: 56

```
0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 ==>
```

Screen: 59

```
0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->
```

Screen: 57

```
0 ( Dillo: reserved )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->
```

Screen: 60

```
0 ( Dillo: POSIT SCRVID )
1
2 FORTH DEFINITIONS
3
4 : POSIT ( xabs yabs -- )
5 90 C! 91 ! ;
6
7
8 '( SCRVID )(
9 : SCRVID
10 559 C@ 252 AND OR 559 C! ; )
11 ( width -- )
12
13
14
15 ==>
```

Screen: 61

```

0 ( Turtle: TRGTBL )
1 DILLO DEFINITIONS
2 '( TABLE TABLE TRGTBL )
3 ( DTBL TRGTBL ) 0 ,
4 571 , 1143 , 1714 , 2285 ,
5 2855 , 3425 , 3993 , 4560 ,
6 5125 , 5689 , 6252 , 6812 ,
7 7370 , 7927 , 8480 , 9031 ,
8 9580 , 10125 , 10667 , 11206 ,
9 11742 , 12274 , 12803 , 13327 ,
10 13847 , 14364 , 14875 , 15383 ,
11 15885 , 16383 , 16876 , 17363 ,
12 17846 , 18323 , 18794 , 19259 ,
13 19719 , 20173 , 20620 , 21062 ,
14 21497 , 21925 , 22347 , 22761 ,
15 -->

```

Screen: 64

```

0 ( Dillo: +-SIN, COS #S/C )
1
2 : +-SIN ( Inl -- n )
3 DAZM 180 )
4 IF MINUS ENDIF ;
5
6 : +-COS ( Inl -- n )
7 DAZM 90 ) DAZM 270 ( AND
8 IF MINUS ENDIF ;
9
10 : #S/C ( -- index )
11 DAZM DUP 180 )
12 IF 180 - ENDIF
13 DUP 90 )
14 IF 180 SWAP- ENDIF ;
15 ==>

```

Screen: 62

```

0 ( Turtle: TRGTBL )
1
2 23169 , 23570 , 23964 , 24350 ,
3 24729 , 25100 , 25464 , 25820 ,
4 26168 , 26509 , 26841 , 27165 ,
5 27480 , 27787 , 28086 , 28377 ,
6 28658 , 28931 , 29195 , 29450 ,
7 29696 , 29934 , 30162 , 30381 ,
8 30590 , 30790 , 30981 , 31163 ,
9 31335 , 31497 , 31650 , 31793 ,
10 31927 , 32050 , 32164 , 32269 ,
11 32363 , 32448 , 32522 , 32587 ,
12 32642 , 32687 , 32722 , 32747 ,
13 32762 , 32767 ,
14
15 ==>

```

Screen: 65

```

0 ( Dillo: COSASP *SIN *COS )
1
2 : COSASP ( dely -- fixed )
3 COSFAC 1000 */ ;
4
5 : *SIN ( n -- n*"sin" )
6 #S/C TRGTBL @ +-SIN
7 32767 */ ;
8
9 : *COS ( n -- n*"cos" )
10 #S/C 90 SWAP- TRGTBL @ +-COS
11 32767 */ CASP ;
12
13
14
15 -->

```

Screen: 63

```

0 ( Dillo: AZMADJ )
1
2 : AZMADJ ( -- )
3 DAZM ABS 360 )=
4 IF DAZM 360 MOD TO DAZM ENDIF
5 DAZM 0(
6 IF 360 AT DAZM +! ENDIF ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 66

```

0 ( Dillo: ASPECT )
1
2 FORTH DEFINITIONS
3
4 : ASPECT DILLO ( ON/OFF -- )
5 IF [ ' COSASP CFA ]L
6 ELSE [ ' NOOP CFA ]L
7 ENDIF
8 TO CASP ;
9
10 OFF ASPECT
11
12
13
14
15 ==>

```

Screen: 67

```

0 ( Dillo: [XX] [YY] DXF DXL )
1 DILLO DEFINITIONS
2
3 : (XX)          ( XnD -- xabs )
4   HSCL + WNDRGT WNDLFT -
5   HSCL 2* NOOP */ WNDLFT + ;
6
7 : (YY)          ( YnD -- yabs )
8   VSCL SWAP- WNDBOT WNDTOP -
9   VSCL 2* NOOP */ WNDTOP + ;
10
11 : DXF ( XnD YnD -- xabs yabs )
12   (YY) SWAP (XX) SWAP ;
13
14 : DXL ( XnD YnD -- xabs yabs )
15   >R HABS + R> VABS SWAP- ; -->

```

Screen: 68

```

0 ( Dillo: <[XX] <[YY] <DXF <DXL )
1
2 : <(XX)          ( xabs -- YnD )
3   WNDLFT - HSCL 2* 1+
4   WNDRGT WNDLFT - */ HSCL - ;
5
6 : <(YY)          ( yabs -- XnD )
7   WNDTOP - VSCL 2* 1+
8   WNDTOP WNDBOT - */ VSCL + ;
9
10 : <DXF ( xabs yabs -- X# Y# )
11   <(YY) SWAP <(XX) SWAP ;
12
13 : <DXL ( xabs yabs -- X# Y# )
14   >R HABS - R> VABS SWAP- ;
15                                     ==>

```

Screen: 69

```

0 ( Dillo: XFORM )
1
2 FORTH DEFINITIONS
3
4 : XFORM DILLO ( f -- )
5   IF [ ' DXF CFA ]L
6     [ ' <DXF CFA ]L 1
7   ELSE [ ' DXL CFA ]L
8     [ ' <DXL CFA ]L 0
9   ENDIF
10  TO ?XFM TO <XF/L TO XF/L ;
11
12 OFF XFORM
13
14
15                                     -->

```

Screen: 70

```

0 ( Dillo: ?HZONE ?VZONE DLINE )
1
2 DILLO DEFINITIONS
3
4 : ?HZONE          ( coor -- f )
5   DUP WNDW (
6   IF DROP -1 ELSE WNDE ) ENDIF ;
7
8 : ?VZONE          ( coor -- f )
9   DUP WNDS (
10  IF DROP -1 ELSE WNDN ) ENDIF ;
11
12 : DLINE          ( -- )
13   X1D Y1D XF/L POSIT
14   X2D Y2D XF/L DRAWLN ;
15                                     ==>

```

Screen: 71

```

0 ( Dillo: HWSTP VWSTP ICLIP )
1
2 : HWSTP 0)          ( f -- n n )
3   IF WNDE ELSE WNDW ENDIF DUP ;
4
5 : VWSTP 0)          ( f -- n n )
6   IF WNDN ELSE WNDS ENDIF DUP ;
7
8 : ICLIP          ( -- )
9   DX1 TO X1D DY1 TO Y1D
10  DX2 TO X2D DY2 TO Y2D
11  DX1 ?HZONE TO IX1
12  DX2 ?HZONE TO IX2
13  DY1 ?VZONE TO IY1
14  DY2 ?VZONE TO IY2 ;
15                                     -->

```

Screen: 72

```

0 ( Dillo: CLIP )
1
2 : CLIP          ( -- )
3   1 ICLIP IX1 IY1 IX2 IY2
4   OR OR OR DUP NOT TO DFLG
5   IX1 IX2 < IX1 0= OR
6   IY1 IY2 < IY1 0= OR AND AND
7   IF IX1
8     IF DY1 DY2 - IX1 HWSTP TO X1D
9     DX2 - DX1 DX2 - */
10    DY2 + DUP ?VZONE
11    IF 2DROP 0
12    ELSE TO Y1D 0 TO IY1
13    1 TO DFLG
14    ENDIF
15    ENDIF
16                                     ==>

```

Screen: 73

```

0 ( Dillo: CLIP )
1
2 IY1
3 IF DX1 DX2 - IY1 VWSTP TO Y1D
4 DY2 - DY1 DY2 - */
5 DX2 + DUP ?HZONE
6 IF 2DROP 0
7 ELSE TO X1D 1 TO DFLG
8 ENDIF
9 ENDIF DFLG OR
10
11 IX2 OVER AND
12 IF DY2 DY1 - IX2 HWSTP TO X2D
13 DX1 - DX2 DX1 - */
14 DY1 + DUP ?VZONE
15 -->

```

Screen: 74

```

0 ( Dillo: CLIP )
1 IF DROP
2 ELSE TO Y2D 0 TO IY2
3 1 TO DFLG
4 ENDIF
5 ENDIF
6 IY2 AND
7 IF DX2 DX1 - IY2 VWSTP TO Y2D
8 DY1 - DY2 DY1 - */
9 DX1 + DUP ?HZONE
10 IF DROP
11 ELSE TO X2D 1 TO DFLG
12 ENDIF
13 ENDIF
14 ELSE DROP
15 ENDIF ; ==>

```

Screen: 75

```

0 ( Dillo: DLN/UPD MOVE TURN[TO])
1 : DLN/UPD ( -- )
2 DFLG IF DLINE ENDIF
3 DY2 TO DY1 DX2 TO DX1 ;
4
5 : MOVE ( n -- )
6 DUP *SIN DX1 + TO DX2
7 *COS DY1 + TO DY2 0 TO DFLG
8 ?DOWN IF CLIP ENDIF DLN/UPD ;
9
10 FORTH DEFINITIONS
11 : TURN DILLO ( azim -- )
12 AT DAZM +! AZMADJ ;
13
14 : TURNTD DILLO ( azim -- )
15 TO DAZM AZMADJ ; -->

```

Screen: 76

```

0 ( Dillo: GO[TO] DUPGO DRAW )
1
2 : GO DILLO ( n -- )
3 ?DOWN SWAP 0 TO ?DOWN MOVE
4 TO ?DOWN ;
5
6 : GOTO DILLO ( x y -- )
7 TO DY1 TO DX1 ;
8
9 : DUPGO ( n -- n )
10 DUP GO ;
11
12 : DRAW DILLO ( n -- )
13 ?DOWN SWAP 1 TO ?DOWN MOVE
14 TO ?DOWN ;
15 ==>

```

Screen: 77

```

0 ( Dillo: DUPDRAW DRAWTO PHPEN )
1
2 : DUPDRAW ( n -- n )
3 DUP DRAW ;
4
5 : DRAWTO DILLO ( x y -- )
6 TO DY2 TO DX2 CLIP DLN/UPD ;
7
8 : PHPEN DILLO ( b -- )
9 TO PHCLR ADJCLR ;
10
11
12
13
14
15 -->

```

Screen: 78

```

0 ( Dillo: GO[TO]. DUPGO. DOT )
1
2 : GOTO. DILLO ( x y -- )
3 2DUP 2DUP GOTO
4 TO DY2 TO DX2 CLIP DFLG
5 IF [ DRWSTT 9 + ]L >R R @
6 0 R ! >R DRAWTO R> R> !
7 ELSE 2DROP
8 ENDIF ;
9
10 : GO. DILLO ( n -- )
11 GO DX1 DY1 GOTO. ;
12
13 : DUPGO. ( n -- )
14 DUP GO. ;
15 ==>

```


Screen: 79

```

0 ( Dillo: DOT PEN DRBAK PHBAK )
1
2 : DOT ( -- )
3 0 GO. ;
4
5 : PEN DILLO ( b -- )
6 DUP TO DRCLR PHPEN ;
7
8 : DRBAK DILLO ( color -- )
9 TO DRUCLR ADJCLR ;
10
11 : PHBAK DILLO ( color -- )
12 DUP TO PHUCLR DRBAK ;
13
14
15 -->

```

Screen: 80

```

0 ( Dillo: CENTER[0] PHIL[TO] )
1
2 : CENTER ( -- )
3 0 0 GOTO ;
4
5 : CENTER0 ( -- )
6 CENTER 0 TO DAZM ;
7
8 : PHIL DILLO ( n -- )
9 DRWSTT C@ -1 DRWSTT C!
10 SWAP DRAW DRWSTT C! ;
11
12 : PHILTO DILLO ( x y -- )
13 DRWSTT C@ -1 DRWSTT C!
14 <ROT DRAWTO DRWSTT C! ;
15 ==>

```

Screen: 81

```

0 ( Dillo: WABS WNDREL WNUM )
1
2 DILLO DEFINITIONS
3
4 : WABS ( 1 r t b -- L R T B )
5 DILLO ROT SWAP XF/L 2SWAP
6 XF/L 2SWAP ROT SWAP ;
7
8 : WNDREL ( 1 r t b -- L R T B )
9 DILLO ROT <XF/L 2SWAP
10 <XF/L 2SWAP ROT SWAP ;
11
12 : WNUM ( 1 r t b -- )
13 DILLO TO WNDL TO WNDN
14 TO WNDL TO WNDW ;
15 -->

```

Screen: 82

```

0 ( Dillo: WEDG WRL,BT- BASNUM )
1
2 : WEDG ( 1 r t b -- )
3 DILLO TO WNDL TO WNDTOP
4 TO WNDRGT TO WNDLFT ;
5
6 : WRL- ( -- n )
7 WNDL WNDL - 2/ ;
8
9 : WBT- ( -- n )
10 WNDL WNDT - 2/ ;
11
12 : BASNUM ( -- )
13 WRL- DUP MINUS SWAP
14 WBT- DUP MINUS WNUM ;
15 ==>

```

Screen: 83

```

0 ( Dillo: SCLSTP RELOC )
1
2 : SCLSTP ( -- )
3 WRL- TO HSCL WBT- TO VSCL ;
4
5 : RELOC ( -- )
6 91 @ 90 C@ <XF/L GOTO ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 84

```

0 ( Dillo: M,SCOOR SET-SCALE )
1
2 : MCOOR DILLO ( -- )
3 SCLSTP 1 TO ?MCOOR
4 OFF XFORM ; MCOOR
5
6 : SCOOR DILLO ( -- )
7 HUSR TO HSCL VUSR TO VSCL
8 HUSR DUP MINUS SWAP
9 VUSR DUP MINUS WNUM
10 0 TO ?MCOOR ON XFORM ;
11
12 : SET-SCALE ( hsc1 vscl -- )
13 DILLO TO VUSR TO HUSR ;
14
15 ==>

```

Screen: 85

```

0 ( Dillo: COOR DSCALE WNDASP WCTR)
1
2 : COOR DILLO ( -- )
3 ?MCOOR
4 IF MCOOR ELSE SCOOR ENDIF ;
5
6 : DSCALE ( -- )
7 WRL- WBT- SET-SCALE ;
8
9 : WNDASP ( -- )
10 CASP SWAP CASP SWAP ;
11
12 FORTH DEFINITIONS
13 : WCTR DILLO ( -- )
14 WNDE WNDW - 2/ WNDW + WNDN
15 WNDS - 2/ WNDS + GOTO ; -->

```

Screen: 86

```

0 ( Dillo: WCTR0 BASWND WINDOW)
1 : WCTR0 ( -- )
2 WCTR 0 TURNT0 ;
3
4 : BASWND DILLO ( -- )
5 WNDL TO WNDLFT WNDR TO WNDRGT
6 WNDT TO WNDTOP WNDB TO WNDBOT
7 BASNUM CENTER0 ;
8
9 : WINDOW ( lft rgt top bot -- )
10 DILLO WNDASP MCOOR 2OVER 2OVER
11 BASWND WABS WEDG WNUM WCTR0 ;
12
13 : RELWND ( lft rgt top bot -- )
14 DILLO WNDASP SCOOR WABS
15 WEDG CENTER0 ; ==>

```

Screen: 87

```

0 ( Dillo: RELWND FRAME )
1
2
3 : FRAME DILLO ( -- )
4 [ DRWSTT 8 + ]L
5 DUP @ OVER @ SWAP ! SWAP
6 WNDLFT WNDTOP POSIT
7 WNDRGT WNDTOP DRAWLN
8 WNDRGT WNDBOT DRAWLN
9 WNDLFT WNDBOT DRAWLN
10 WNDLFT WNDTOP DRAWLN ! ;
11
12
13
14
15 -->

```

Screen: 88

```

0 ( Dillo: DINIT WIPE )
1
2 : DINIT DILLO ( -- )
3 DRWSTT 16 ERASE
4 ON RPHIL
5 MCOOR BASWND 0 PHBAK 1 PEN ;
6 DINIT
7
8 : WIPE DILLO ( org dest -- )
9 [ DRWSTT 9 + ]L >R R @ 0 R !
10 >R PHCLR DRCLR PHUCLR PEN
11 WNDBOT 1+ WNDTOP
12 DO WNDLFT I POSIT
13 WNDRGT I DRAWLN
14 LOOP PEN PHPEN R> R> ! ;
15 ==>

```

Screen: 89

```

0 ( Dillo: ASPSTP )
1
2 DILLO DEFINITIONS
3
4 : ASPSTP ( tbladr -- )
5 DUP C@ SWAP 1+ C@ * 32 SWAP
6 / 833 * TO COSFAC ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 90

```

0 ( Dillo: DEFBAS EDGES )
1
2 : DEFBAS ( L R T B -- )
3 TO WNDB TO WNDT
4 TO WNDR TO WNDL
5 WRL- TO HABS WBT- TO VABS
6 SCLSTP ;
7
8 : EDGES ( scr/pxl pxl/ln -- )
9 1- ( WNDR ) >R
10 703 C@ 4 = ( ?split )
11 IF 160 ELSE 192 ENDIF
12 SWAP / 1- ( WNDB )
13 0 R> 0 4 ROLL DEFBAS
14 BASWND ;
15 ==>

```

Screen: 91

```

0 ( Dillo:  PXLTBL
1 LABEL PXLTBL
2 ( scn/pxl  pxl/ln/10  pxl/byt  )
3   8 C,      4 C,      1 C,      ( 0 )
4   8 C,      2 C,      1 C,      ( 1 )
5  16 C,      2 C,      1 C,      ( 2 )
6   8 C,      4 C,      4 C,      ( 3 )
7   4 C,      8 C,      8 C,      ( 4 )
8   4 C,      8 C,      4 C,      ( 5 )
9   2 C,      16 C,     8 C,      ( 6 )
10  2 C,      16 C,     4 C,      ( 7 )
11  1 C,      32 C,     8 C,      ( 8 )
12  1 C,      8 C,      2 C,      ( 9 )
13  1 C,      8 C,      2 C,      ( 10 )
14  1 C,      8 C,      2 C,      ( 11 )
15  1 C,      16 C,     4 C,      -->

```

Screen: 92

```

0 ( Dillo:  7PLUS
1
2 CODE 7PLUS HEX      ( -- )
3   A9 C, 07 C, 85 C, 57 C, AD C,
4   30 C, 02 C, 85 C, N C, AD C,
5   31 C, 02 C, 85 C, N 1+ C, HERE
6   B1 C, N C, 29 C, FC C,
7   C9 C, 40 C, F0 C,
8   14 C, B1 C, N C,
9   85 C, N 2+ C, 29 C,
10  0F C, C9 C, 0F C, D0 C, 06 C,
11  C6 C, N 2+ C, A5 C, N 2+ C,
12  91 C, N C, C8 C, 4C C, ,
13  4C C, NEXT, C; DCX
14
15 ==>

```

Screen: 93

```

0 ( Dillo:  DIMSTP
1
2 : DIMSTP      ( stripped-GR-# -- )
3               ( scn/pxl pxl/ln  )
4   3 * PXLTBL + DUP ASPSTP
5   DUP C@ ( scn/pxl ) SWAP
6   DUP 2+ C@ TO PX/BYT
7   1+ C@ 10 * ( pxl/ln, normal )
8   559 ( DMACTL ) C@
9   3 AND DUP 2 <>
10  IF 1 = IF 4 ELSE 6 ENDIF 5 */
11  ELSE DROP
12  ENDIF ( pxl/ln, actual )
13  DUP PX/BYT / TO BYT/LN
14
15 -->

```

Screen: 94

```

0 ( Dillo:  DIMSTP  UGR.
1
2   PX/BYT DUP 2 =
3   IF DROP 12
4   ELSE 4 =
5   IF 8 ELSE 0 ENDIF
6   ENDIF
7   MSKTBL + MTBL1 8 CMOVE
8   8 0
9   DO
10    MTBL1 I + C@
11    255 XOR MTBL2 I + C!
12  LOOP ;
13
14 : UGR.      ( stripped-GR-# -- )
15  DIMSTP 2DROP MCOOR ;      ==>

```

Screen: 95

```

0 ( Dillo:  GR.
1
2 FORTH DEFINITIONS
3
4 : GR. DILLO      ( b -- )
5   ( set up display, 7+ ? )
6   DUP 15 AND SWAP OVER
7   12 =
8   IF 2- 2- GR. 7PLUS
9   ELSE GR.
10  ENDIF
11  ( set up drawln & window data)
12  DIMSTP EDGES
13  ( initialization )
14  0 PHBAK 1 PEN MCOOR DSCALE
15  OFF ASPECT ; FORTH

```

Screen: 96

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 97

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 98

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 99

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 100

0 (turntwd: TURNTWD)
1
2 ' (QATN2) (17 KLOAD)
3
4 : TURNTWD DILLO (x y --)
5 DY1 - SWAP DX1 - SWAP
6 QATN2 DUP ABS 87 + SWAP +-
7 (ROUNDING)
8 180 31416 */
9 (DEGREES)
10 90 SWAP- TURNTWO ;
11
12
13
14
15

Screen: 101

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 102

0 (P-naming: NAMEPT THISPT)
1
2 : NAMEPT (xxx, x y --)
3 (BUILDS , , (xxx: -- x y)
4 DOES> DUP 2+ @ SWAP @ ;
5
6 : THISPT DILLO (xxx, --)
7 (xxx: -- x y)
8 DX1 DY1 NAMEPT ;
9
10 (or write 2QUAN)
11
12
13
14
15

Screen: 103

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 106

0 (lines: 2LNX MAKLN)
1
2 (a1 b1 c1 a2 b2 c2 -- x y)
3
4 5 PICK 2 PICK M* (b1*c2)
5 4 PICK 7 PICK M* (- b2*c1)
6 DMINUS D+
7 I' J D/ DROP >R (x)
8 2DROP 2DROP 2DROP
9 R> R> R> R> 2DROP ;
10
11 : MAKLN DILLO (-- a b c)
12 DX1 DY1
13 OVER 100 *SIN +
14 OVER 100 *COS + 2PT-LN ;
15

Screen: 104

0 (lines: 2PT-LN)
1 DILLO DEFINITIONS QUAN LNA
2 QUAN LNB FORTH DEFINITIONS
3 : 2PT-LN (x1 y1 x2 y2 --a b c)
4 DILLO 2DUP >R >R
5 ROT - TO LNA (dy = a)
6 - TO LNB (-dx = b)
7 LNA R> M* (-ax2-by2)
8 LNB R> M* D+ DMINUS (= c)
9 2DUP 32767 M/ ABS
10 SWAP DROP -DUP
11 IF 1+ >R R M/ SWAP DROP
12 LNA R / TO LNA
13 LNB R> / TO LNA
14 ELSE DROP
15 ENDIF LNA LNB ROT ; ==>

Screen: 107

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 105

0 (lines: 2LNX)
1
2 '(D/)(60 KLOAD)
3
4 FORTH DEFINITIONS
5
6 : 2LNX (a1 b1 c1 a2 b2 c2)
7 DILLO (-- x y)
8 6 PICK 3 PICK M* (a1*b2)
9 5 PICK 8 PICK M* (- a2*b1)
10 DMINUS D+ >R >R
11 4 PICK 4 PICK M* (c1*a2)
12 3 PICK 9 PICK M* (- c2*a1)
13 DMINUS D+
14 I I' D/ DROP >R (y)
15 -->

Screen: 108

0 (L-naming: THISLN)
1
2 '(2PT-LN)(52 KLOAD)
3
4 : THISLN (xxx, --)
5 DILLO (xxx: -- a b c)
6 <BUILDS
7 MAKLN , , ,
8 DOES)
9 DUP 4 + @ OVER 2+ @ ROT @ ;
10 (or write 3QUAN)
11
12
13
14
15 ==>

Screen: 109

```

0 ( L-naming: NAMELN )
1
2 : NAMELN ( xxx, x1 y1 x2 y2 -- )
3 (BUILDS ( xxx: -- a b c )
4 2PT-LN , , ,
5 DOES)
6 DUP 4 + @ OVER 2+ @ ROT @ ;
7
8 ( or write 3QUAN )
9
10
11
12
13
14
15

```

Screen: 112

```

0 ( Dillo option load block )
1
2
3 100 LOAD ( TURNTWD )
4
5 102 LOAD ( POINT NAMING )
6
7 104 LOAD ( LINE-INTERSECT. )
8
9 108 LOAD ( LINE NAMING )
10
11 110 LOAD ( WINDOW NAMING )
12
13
14
15

```

Screen: 110

```

0 ( W-naming: THISWND )
1
2 : THISWND DILLO ( xxx, -- )
3 (BUILDS ( xxx: -- )
4 ?XFM ,
5 WNDW , WNDE , WNDN , WNDS ,
6 WNDLFT , WNDRGT ,
7 WNDTOP , WNDBOT ,
8 DOES) >R 18 @
9 DO J I + @ 2 +LOOP R) DROP
10 WEDG WNUM XFORM WCTR@ ;
11
12
13
14
15

```

Screen: 113

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 111

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 114

```

0 ( Quan: TO AT )
1
2 : TO
3 -FIND @= @ ?ERROR DROP
4 STATE @
5 IF ,
6 ELSE EXECUTE
7 ENDIF ; IMMEDIATE
8
9 : AT
10 -FIND @= @ ?ERROR DROP
11 4 + [COMPILE] LITERAL ;
12 IMMEDIATE
13
14
15

```

Screen: 115

```

0 ( Quan: [206] [2!4] )
1
2 ASSEMBLER HEX
3
4 LABEL (206)
5 A0 C, 06 C, B1 C, W C, 48 C,
6 C8 C, B1 C, W C, 4C C, PUSH ,
7
8 LABEL (2!4)
9 A0 C, 04 C, B5 C, 00 C, 91 C,
10 W C, C8 C, B5 C, 01 C, 91 C,
11 W C, 4C C, POP ,
12
13
14
15 -->

```

Screen: 118

```

0 ( Quan: QUAN VECT )
1
2 : QUAN
3 ON PTCH LABEL -2 ALLOT
4 (206) , (2!4) ,
5 [ ' VARIABLE 4 + ] LITERAL ,
6 2 ALLOT OFF PTCH ;
7
8 : VECT
9 ON PTCH LABEL -2 ALLOT
10 (2V4) , (2!2) ,
11 [ ' NOOP CFA ] LITERAL ,
12 OFF PTCH ;
13
14
15

```

Screen: 116

```

0 ( Quan: [2!2] [2V4] )
1
2 LABEL (2!2)
3 A0 C, 02 C,
4 4C C, ' (2!4) 2 + ,
5
6 LABEL (2V4)
7 A0 C, 05 C, B1 C, W C, 48 C,
8 88 C, B1 C, W C, 85 C, W C,
9 68 C, 85 C, W 1+ C,
10 A0 C, 00 C, 4C C, W 1- ,
11
12
13
14
15 ==>

```

Screen: 119

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 117

```

0 ( Quan: patch for CREATE )
1
2 DCX
3
4 : (PTCH) ( system )
5 SWAP >R R = 251 R = 249 R> =
6 OR OR ;
7
8 : PTCH ( system )
9 IF [ ' (PTCH) CFA ] LITERAL
10 ELSE [ ' = CFA ] LITERAL
11 ENDIF
12 [ ' CREATE 63 + ] LITERAL ! ;
13
14
15 -->

```

Screen: 120

```

0 ( Dbls: DU/MOD )
1 DILLO DEFINITIONS
2 HEX
3 CODE DU/MOD ( d1 d2 -- dr dq )
4 A9 C, 04 C, 20 C, SETUP ,
5 CA C, 94 C, 00 C,
6 CA C, 94 C, 00 C,
7 CA C, 94 C, 00 C,
8 CA C, 94 C, 00 C,
9 A0 C, 04 C, CA C, B9 C, C5 C,
10 00 C, 95 C, 00 C, 88 C, D0 C,
11 F7 C, A0 C, 20 C, 16 C, 02 C,
12 36 C, 03 C, 36 C, 00 C, 36 C,
13 01 C, 36 C, 06 C, 36 C, 07 C,
14 36 C, 04 C, 36 C, 05 C, 38 C,
15 ==>

```

Screen: 121

```
0 ( Dbls: DU/MOD )
1
2 B5 C, 06 C, E5 C, C4 C, 48 C,
3 B5 C, 07 C, E5 C, C5 C,
4 48 C, B5 C, 04 C, E5 C, C2 C,
5 48 C, B5 C, 05 C, E5 C, C3 C,
6 30 C, 10 C, 95 C, 05 C, 68 C,
7 95 C, 04 C, 68 C, 95 C, 07 C,
8 68 C, 95 C, 06 C, F6 C, 02 C,
9 4C C, HERE 05 + , 68 C, 68 C,
10 68 C, 88 C, D0 C, C4 C, 4C C,
11 NEXT , C;
12
13 DCX
14
15 -->
```

Screen: 124

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 122

```
0 ( Dbls: D/MOD D/ )
1
2 : D/MOD
3 DUP 4 PICK DUP >R XOR >R
4 DABS 2SWAP DABS 2SWAP
5 DU/MOD R> D+-
6 2SWAP R> D+- 2SWAP ;
7
8 : D/
9 D/MOD 2SWAP 2DROP ;
10
11
12
13
14
15
```

Screen: 125

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 123

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 126

```
0 ( Quan: TO AT )
1
2 : TO
3 -FIND 0= 0 ?ERROR DROP
4 STATE @
5 IF ,
6 ELSE EXECUTE
7 ENDIF ; IMMEDIATE
8
9 : AT
10 -FIND 0= 0 ?ERROR DROP
11 4 + [COMPILE] LITERAL ;
12 IMMEDIATE
13
14
15 ==>
```


Screen: 127

```

0 ( Quan:  QUAN )
1
2 ASSEMBLER HEX
3
4 LABEL (206)
5  A0 C, 06 C, B1 C, W C, 48 C,
6  C8 C, B1 C, W C, 4C C, PUSH ,
7
8 LABEL (2!4)
9  A0 C, 04 C, B5 C, 00 C, 91 C,
10 W C, C8 C, B5 C, 01 C, 91 C,
11 W C, 4C C, POP ,
12
13
14
15 -->

```

Screen: 128

```

0 ( Quan:  [2!2]  [2V4] )
1
2 LABEL (2!2)
3  A0 C, 02 C,
4  4C C, ' (2!4) 2 + ,
5
6 LABEL (2V4)
7  A0 C, 05 C, B1 C, W C, 48 C,
8  88 C, B1 C, W C, 85 C, W C,
9  68 C, 85 C, W 1+ C,
10 A0 C, 00 C, 4C C, W 1- ,
11
12
13
14
15 ==>

```

Screen: 129

```

0 ( Quan:  QUAN  VECT )
1
2 : QUAN
3  LABEL -2 ALLOT
4  (206) , (2!4) ,
5  [ ' VARIABLE 4 + ] LITERAL ,
6  2 ALLOT ;
7
8 : VECT
9  LABEL -2 ALLOT
10 (2V4) , (2!2) ,
11 [ ' NOOP CFA ] LITERAL , ;
12
13
14 DCX
15

```

Screen: 130

```

0 ( Demos:  BOX )
1
2 : BOX ( n -- )
3  4 0
4  DO DUPDRAW 90 TURN
5  LOOP DROP ;
6
7
8
9
10
11
12
13
14
15 -->

```

Screen: 131

```

0 ( Demos:  QUBE )
1
2 : QUBE ( n -- )
3  >R DX1 DY1 DAZM R)
4  DUPDRAW 90 TURN
5  DUPDRAW 45 TURN
6  DUPDRAW 45 TURN
7  DUPDRAW 90 TURN
8  DUPDRAW 45 TURN
9  DUPDRAW 135 TURN
10 DUPDRAW -90 TURN
11 DUPDRAW 180 TURN
12 DUPGO
13 -45 TURN DRAW
14 TURNT0 GOTO ;
15 -->

```

Screen: 132

```

0 ( Demos:  TCIRCLE  QCIRCLE )
1
2 : TCIRCLE ( chord -- )
3  18 0
4  DO DUPDRAW 20 TURN
5  LOOP DROP ;
6
7 : QCIRCLE ( radius -- )
8  DAZM DX1 DY1 4 ROLL
9  -10 TURN
10 3473 10000 */ DUP
11 10000 3473 */ GO
12 100 TURN TCIRCLE
13 GOTO TURNT0 ;
14
15

```

Screen: 133

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 136

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 134

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 137

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 135

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 138

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 139

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 142

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 140

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 143

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 141

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 144

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 145

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 148

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 146

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 149

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 147

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 150

0 (Fp ext: useful constants)
1
2 FP 3.14159265 FCONSTANT PI
3 FP 1.57079633 FCONSTANT PI/2
4 FP 1.04719756 FCONSTANT PI/3
5 FP .785398163 FCONSTANT PI/4
6 FP .523598776 FCONSTANT PI/6
7 FP .0174532925 FCONSTANT RD/DG
8 FP 57.2957796 FCONSTANT DG/RD
9 FP 2.71828183 FCONSTANT EXP1
10 FP 1 FCONSTANT FP1
11 FP 0 FCONSTANT FP0
12 FP 1E+97 FCONSTANT FTOP
13 FP 1E-97 FCONSTANT FBOT
14
15

==>

Screen: 151

```

0 ( Fp ext: FMINUS 2FDUP F+! FMAX)
1
2 : FMINUS          ( fp -- fp )
3   SP@ 4 + DUP C@
4   128 XOR SWAP C! ;
5
6 : 2FDUP ( fp fp -- fp fp fp fp )
7   FOVER FOVER ;
8
9 : F+!           ( fp a -- )
10  DUP >R F@ F+ R) F! ;
11
12 : FMAX          ( fp fp -- fp )
13   2FDUP F<
14   IF FSWAP ENDIF FDROP ;
15                                     -->

```

Screen: 152

```

0 ( Fp ext: FMIN F0< FABS 2FDUP F, )
1 : FMIN          ( fp fp -- fp )
2   2FDUP F>
3   IF FSWAP ENDIF FDROP ;
4
5 : F0<           ( fp -- f )
6   FP0 F< ;
7
8 : FABS          ( fp -- fp )
9   FDUP F0< IF FMINUS ENDIF ;
10
11 : 2FDROP        ( fp fp -- )
12   FDROP FDROP ;
13
14 : F,            ( fp -- )
15   SWAP ROT , , , ;

```

Screen: 153

```

0 ( Fp ext: F>R FR) F.S          )
1 : F>R          ( fp -- )
2   SWAP ROT R) <ROT
3   >R >R SWAP >R >R ;
4
5 : FR)          ( -- fp )
6   R) R) SWAP R) R)
7   ROT >R <ROT SWAP ;
8
9 : F.S          ( -- )
10  CR S@ @ SP@ - 2- 6 / -DUP
11  IF -1 SWAP 1-
12  DO SP@ I 6 * + F@ SWAP ROT F.
13  -1 +LOOP
14  ELSE ." No FP on stack... "
15  ENDIF ;

```

Screen: 154

```

0 ( Fp ext: [F@] FPICK FROLL )
1
2 CODE (F@)      ( system: -- [fp] )
3   HEX CA C, CA C, CA C, CA C,
4   CA C, CA C, 4C C, NEXT , DCX
5
6 : FPICK ( fp..fp n -- fp..fp fp )
7   1 - 6 * SP@ 2+ SWAP 0+S
8   6 - 6 CMOVE (F@) ;
9
10 : FROLL   ( fp..fp n -- fp..fp )
11   0 MAX -DUP
12   IF DUP 6 * >R FPICK SP@
13   DUP 6 + R) <CMOVE FDROP
14   ENDIF ;
15                                     ==>

```

Screen: 155

```

0 ( Fp ext: -FIX -FLOAT          )
1
2 : -FIX          ( fp -- n )
3   FP -32768 FMAX FP 32767 FMIN
4   FDUP F0<
5   IF FMINUS 1 ELSE 0 ENDIF
6   >R FIX R)
7   IF MINUS ENDIF ;
8
9 : -FLOAT        ( n -- fp )
10  DUP 0< DUP >R
11  IF MINUS ENDIF
12  FLOAT R)
13  IF FMINUS ENDIF ;
14
15                                     -->

```

Screen: 156

```

0 ( Fp ext: FLWCHK [system]      )
1
2 : FLWCHK        ( system )
3   DUP 173 )      ( fp fp n n n -- )
4   IF DROP 128 ( fp fp 1 / fp 0 )
5   AND SWAP 128 AND
6   XOR >R 2FDROP
7   FTOP R)
8   IF FMINUS
9   ENDIF 0
10  ELSE 79 (
11  IF 2DROP 2FDROP FP0 0
12  ELSE 2DROP 1
13  ENDIF
14  ENDIF ;
15                                     ==>

```

Screen: 157

```

0 ( Fp ext: 2EXP0& [system] )
1
2 : 2EXP0& ( system )
3 ( fp fp -- fp fp n n n )
4 SP0 10 + DUP C0 SWAP 6 - C0
5 2DUP 127 AND SWAP
6 127 AND SWAP ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 158

```

0 ( Fp ext: F*OV F/OV 1/FP )
1
2 : F*OV ( fp fp -- fp )
3 2EXP0& + FLWCHK IF F* ENDIF ;
4
5 : F/OV ( fp fp -- fp )
6 FDUP F0=
7 IF FDROP FBOT
8 ENDIF
9 2EXP0& 128 - MINUS + FLWCHK
10 IF F/
11 ENDIF ;
12
13 : 1/FP ( fp -- fp )
14 FP1 FSWAP F/OV ;
15 ==>

```

Screen: 159

```

0 ( Fp ext: ->RD ->DG )
1
2 : FR ( -- fp1 )
3 4 RPICK J I' ;
4
5 : ->RD ( fp1 -- fp2 )
6 RD/DG F* ;
7
8 : ->DG ( fp1 -- fp2 )
9 DG/RD F* ;
10
11
12
13
14
15

```

Screen: 160

```

0 ( Trig: sin/cos & atn coefs )
1
2 ' ( 1/FP ) ( 15 KLOAD )
3
4 LABEL SCCFS
5 FP .260190304E-5 F,
6 FP -.198074187E-3 F,
7 FP .833302514E-2 F,
8 FP -.166666567 F,
9
10 LABEL TCFS
11 FP -.509095825E-1 F,
12 FP -.470832514 F,
13 FP .141250074E+1 F,
14
15 ==>

```

Screen: 161

```

0 ( Trig: [S/C] )
1
2 : (S/C) ( fp f -- fp )
3 >R FDUP PI F/ FIX DUP 1 AND
4 IF R> 1 XOR >R
5 ENDIF
6 FLOAT PI F* F-
7 FDUP FDUP F* FDUP F>R
8 SCCFS 4 FR> FS FPOLY >F F*
9 FOVER F* F+
10 R>
11 IF FMINUS
12 ENDIF ;
13
14
15 -->

```

Screen: 162

```

0 ( Trig: SIN COS TAN )
1
2 : SIN ( fp -- fp )
3 FDUP F0< >R
4 FABS R> (S/C) ;
5
6 : COS ( fp -- fp )
7 FABS PI/2 F+
8 0 (S/C) ;
9
10 : TAN ( fp -- fp )
11 FDUP SIN FSWAP COS F/OV ;
12
13
14
15 ==>

```

Screen: 163

```

0 ( Trig: [ATN] )
1
2 : (ATN) ( fp -- fp )
3 FDUP FP1 F)
4 IF 1/FP 2 ELSE 0 ENDIF >R
5 FDUP FP .267949192 F)
6 IF FDUP FP 1.73205081 F* FP1
7 F- FSWAP FP 1.73205081
8 F+ F/ R) 1+ >R
9 ENDIF
10 FDUP FDUP F* FDUP FDUP F>R
11 TCFS 2 FR) FS FPOLY >F F*
12 FSWAP TCFS 12 + F@ F+ F/
13 FOVER F* F+
14 I 1 )
15 -->

```

Screen: 166

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 164

```

0 ( Trig: [ATN] )
1
2 IF FMINUS
3 ENDIF R) -DUP
4 IF
5 1- -DUP
6 IF
7 1-
8 IF PI/3
9 ELSE PI/2
10 ENDIF
11 ELSE PI/6
12 ENDIF F+
13 ENDIF ;
14
15 ==>

```

Screen: 167

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 165

```

0 ( Trig: ATN ATN2 )
1
2 : ATN ( fp -- fp )
3 FDUP F0( >R FABS (ATN) R)
4 IF FMINUS
5 ENDIF ;
6
7 : ATN2 ( fpx fpy -- fp )
8 FSWAP 2FDUP FSWAP
9 F0( >R F0( >R
10 F/OV FABS (ATN) R)
11 IF PI FSWAP F-
12 ENDIF R)
13 IF FMINUS
14 ENDIF ;
15

```

Screen: 168

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 169

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 172

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 170

0 CONTENTS OF THIS DISK:

1
2 ARMADILLO GRAPHICS: 40 LOAD
3 + TURN TOWARD: 100 LOAD
4 + POINT-NAMING: 102 LOAD
5 + LINE-INTERSECT FCNS: 104 LOAD
6 + LINE-NAMING: 108 LOAD
7 + WINDOW-NAMING: 110 LOAD
8 LOAD ALL ABOVE +'S: 112 LOAD
9 DEMOS: 130 LOAD
10
11 FLOATING PT. EXTENSION: 150 LOAD
12 (REQUIRES VAL4TH FLT. PT.)
13 QTRIG (INTEGER) FNCTNS: 30 LOAD
14
15 QUAN STRUCTURES: 114 LOAD

Screen: 173

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 171

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 179

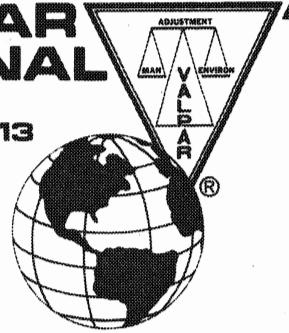
0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

**VALPAR
INTERNATIONAL**

3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141



valFORTH^{T.M.}
SOFTWARE SYSTEM
for ATARI*

**Text Compression and
Auto Text Formatting**

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

Software and Documentation
© Copyright 1982
Valpar International

valFORTHTM
SOFTWARE SYSTEM

Text Compression and Auto Text Formatting

Evan Rosen

Software and Documentation

© Copyright 1982

Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a \$12.00 Replacement Fee.

TEXT COMPRESSION AND AUTO TEXT FORMATTING

Table of Contents

LI	OVERVIEW AND STROLLING THROUGH TCAF
LII	TCAF - GLOSSARY
LIII	TCAF - SUPPLIED SOURCE LISTING

NOTICE
TEXT COMPRESSION AND AUTOMATIC TEXT FORMATTING
CODE TRANSPORTATION

The routines in this package have been coded and presented so that they may be readily transported to other fig-FORTH systems on machines other than the Atari 400/800. This is in response to numerous requests to this effect from various "adventure" game authors. We note, however, that the same restrictions apply to the software in this package, whether run on the Atari 400/800 machines or any other:

First, the code may ONLY be used in either an AUTO'd system as described in valFORTH 1.1 documentation, or in a target-compiled system.

Second, any software written with these routines, on any machine, must contain the acknowledgement of Valpar International as the source of the code, as described and detailed in valFORTH 1.1 documentation.

Other distribution may be construed to be a violation of applicable copyright laws.

Overview

This package attempts to fill at least two common needs of the programmer who does verbal/interactive programming.

First, a group of automatic text formatting routines is provided that allow two different approaches:

- *A non-wrap line formatter to both the video display and the printer, including variable-margin capability and inverse video option, and
- *A versatile window formatting system, with scrolling, color and inverse video options as appropriate, and window naming. Notes on the creation of window types with different "generic" parameters are also included.

In both modes described above, user options of left-, right-, center-, or fill-justification are supported, as is numerical output formatting.

Second, two different approaches to the problem loosely termed "text compression" are implemented:

The first is intended for use in programs where run-time retrieval of text stored on disk is allowable, and provides a set of general virtual-memory operators for the creation and retrieval of messages from disk. A simple encryption scheme is provided as a (working) example for the software developer who wishes his or her messages to be not easily readable from disk with, for instance, a Forth screen editor. In addition, alternate points in the virtual memory routines are indicated where deeper encryption routines might be employed. Routines are provided for virtual memory message programming on both one- and two-drive development systems.

The second set of text compression routines is intended for use in "in memory" applications, such as cassette-booted programs, that do not have access to disk for message retrieval. In this case the most compact code practical is desired, and a system built around some of the basic aspects of Forth's compact threaded-code structure is provided.

Finally, we note that the autoformatting and text compression utilities are designed to be used in most any of their possible different combinations.

STROLLING THROUGH TCAF (Text Compression and Autoformatting)

The organization of this disk is slightly different from the others in this series. While a table of contents may still be found on screen 170 as usual, in this package the "load chain," starting on screen 166, will get far more use. In general when one wishes to load a TCAF development system with a specific set of capabilities, one makes slight adjustments to the load chain option screen and then simply loads the first screen in the chain. The chain does the rest.

To start off, first prepare two blank, formatted disks. Make your normal working copy of TCAF on the first disk and leave it un-write-protected. The second disk will be used a little later.

Autoformatting

In order to select options you will want to make changes to the load chain option screen. This may be found by locating the load chain in the directory on screen 170, and then scanning through the screens in the chain until you find the one marked "options" in its first line. (This is on or near screen 167.) Look at this screen, and see that most of the lines have a left parenthesis in the left column, followed by a LOAD command and a comment. By removing selected left-column left parentheses you can activate various options. Right now on your working copy use an editor to remove all of the left-column left parentheses except for the one on the line that says "text compression." (Text compression uses transient structures and will be discussed separately.) And, of course, don't remove the one in the comment at the very top of the screen. OK, now boot a bare valFORTH 1.1 system, and load in the debugger, and swap in the TCAF disk, do MTB as usual, and load the first screen in the load chain on this disk. (Probably 166.)

When the prompt comes back, type

ON STACK

since you'll want to watch the stack. Then type

TYPEOUT

(Failure to execute this initialization word may cause a crash as you try to use words like *TYPE later on.) This command activates one of the two formatting modes. This mode, called "type-out mode," since it uses the word TYPE as its actual output word, can send formatted type to either the display or the printer. The other formatting mode is activated by WINDOWOUT and is called "window-out mode." It will be discussed a bit later.

Now type

```
" Here is a simple example of the formatter's function."
```

using lower case as shown, and notice that an address, actually PAD, is left on the stack. Now reactivate upper case (press Shift and Caps-Lowr) if you haven't already, and type

```
COUNT
```

The address was bumped by one, and the string count was extracted from the first byte in the string created by " and placed on top of stack. All normal. Now type

```
2DUP CR CR TYPE CR
```

and see that the typed output wraps around as usual. Now try

```
2DUP CR CR *TYPE *CR
```

The word "formatter's" is no longer split. Let's try it again but with different formatting. Type

```
CTRJST ("center justification")  
2DUP CR CR *TYPE *CR
```

How about

```
FILJST ("fill justification")  
2DUP CR CR *TYPE *CR
```

The text is now spread or "filled" to take up the whole space between the margins. The last mode is

```
RG TJST ("right justification")  
2DUP CR CR *TYPE *CR
```

which gives the expected result. Finally, type

```
LFTJST ("left justification," the default mode)  
2DUP CR CR *TYPE *CR
```

and we're back where we started.

Well, what precisely is happening? The 2DUP each time is there of course to reproduce the two stack arguments, adress and count, for use by *TYPE (or TYPE). The two CR's each time are merely to space the result down the page a bit, and make it start at the left margin. As we will see, these two CR's will not generally be necessary in normal programs. The TYPE we'll assume you already know about. If not, look it up in the 1.1 glossary. While you're looking at TYPE's definition you might refresh your memory about how to allow it to type inverse video characters also. A short discussion about this follows TYPE's definition, and we may need this feature later on. OK, what about *TYPE? *TYPE, like TYPE, takes a count and address on the stack, but instead of routing the text directly to an output device, *TYPE sends it instead to a

holding buffer, located at BUF, where it accumulates. As each character is sent to the buffer it may be colored, inversed, or capitalized, depending on whether these options are loaded and appropriate. Since we loaded all three of these options we'll try them presently. When the buffer at BUF overflows with a non-blank character, *TYPE formats the line (if any format routines were loaded) and then sends it out via a vect called *XMTLN. Roughly, a vect is a word that can be "assigned" the meaning of a second word so that when the vect is executed it acts precisely like the word last assigned to it.) "XMTLN" in *XMTLN stands for "transmit line." The word *XMTLNP is currently assigned to *XMTLN and is located, in the first release, on or near screen 73. *XMTLNP, and so now *XMTLN, types out the buffer at BUF and increments a line counter if the printer is on and does a few CR's if a printed page is full. After the buffer is output, cleared, and the overhanging characters have been moved to the beginning of the buffer, *TYPE continues to consume the character string. In general there will always be something in the buffer unless it has been cleared. *CR pushes the last of the text out of the buffer to the output device, and then clears and initializes the buffer with BUFINIT. You probably won't have to do BUFINIT yourself unless you are experimenting with the internals of the program. To illustrate this point about *CR, type

```
2DUP CR CR *TYPE 2DUP *TYPE *CR
```

This time, since we didn't do *CR after the first *TYPE, the next *TYPE tacked its text right on to what was left in the buffer.

Now type

```
SP! (we'll make a new message)
: SHOW 2DUP CR CR *TYPE *CR ;
" here is another message for another purpose."
COUNT
CTRJST
CAP SHOW
SHOW
```

See what CAP does? Now try

```
ON CAPS
SHOW
OFF CAPS
SHOW
```

And what about inverse video? Since *TYPE uses TYPE and TYPE as it now stands will not print inverse video (it strips the high bit before sending a byte out) we'll need the modification discussed in the 1.1 Glossary, under TYPE. Here it is, type it in, c a r e f u l l y:

```
HEX FF ' TYPE 14 + C! DECIMAL
```

and then type

```
VLIST
```

to see an interesting side note. The high bits of the last byte of (almost) all

names are set. This is used by vocabulary search words. But, back to business.
Type

SHOW

And you get trash. This is because some word in VLIST uses PAD for something,
so your message was overwritten. This is just a reminder. Type

```
SP! (clear the stack)
" Here is yet another message, to show other features."
COUNT
ON CAPS
ON INVID
SHOW
ON INVBK (for "inverse background")
SHOW
OFF INVID
SHOW
2DUP CR *CR *TYPE *CR *CR CR
etc.
```

Play with these things for a while if you like. When you're done, do

```
SP!
OFF CAPS
OFF INVID
OFF INVBK
```

and we'll continue.

Virtual Memory (Disk-stored) Text

We used the word " in the exercises above, but it has a serious shortcoming in that it won't digest a string larger than 255 characters since it only keeps a one-byte length byte. The word X" (for "extended quote") in this package allows longer strings when loading from disk. X" will not work from the keyboard. Use XCOUNT with X", and it retrieves the two-byte length count and leaves it on top of the stack. There is a demo of X" on screen 120. Take a look at it if you like and then type

120 LOAD

and a short message will come back as part of the demo. Turn your stack on if you've turned it off for any reason, and do

```
XCOUNT
OFF CAPS OFF INVID OFF INVBK
FILJST
SHOW
```

If you looked at the screen you may have noticed the right-arrow characters near the end of the text. These cause a *CR to be executed at that point in the text. See *EMIT code for details. You can make your own control characters in a similar fashion. (To type a right arrow character in the valFORTH 1.1 editor, do ESC followed by CTRL-*). Observe also that no --> was required for X" to cross the screen boundary. X" will only stop on finding a final " and so may run right on through a disk looking for one if you forget to put it in.

Well, X" is ok, but not as handy as it might be for general programming. Look at screen 122 and then type

```
SP!  
122 LOAD
```

and a demo message will come back again, indicating that a new word, MSGDEM1, now exists. This word will actually pull its message text off the disk. Let's do it. Do MTB just to make sure it's not cheating, and then type

```
CR CR MSGDEM1
```

Notice that we didn't use SHOW this time, just the message name. The messages end with a right-arrow. Now, the method that generates this message, namely using a new word, V" , followed by a string and then a terminating " and then the word M: followed by the message name, does achieve the desired result, but at the price of leaving the V", ", M:, and name on the disk along with the message. This method is provided only because for those working with a one-drive development system it is the easiest, and does not involve any disk swapping during compile time. However, for those with two drive systems, and those with only one drive but also a tolerance for swapping disks every time a message is compiled, the next and last structure in this series is provided. It allows fully compact, text-only messages to be compiled on the final product, and also allows encryption of the text. We will first do it the way the one-drivers need to.

Look at screen 124. The 80 ALTINIT command sets up an alternate set of disk pointers to start at screen 80. This is where, in our example, the text of the various messages compiled by this method will be stored on the extra disk we formatted at the beginning. Notice that the message starts with X" again. Hence, we see that it will first be assembled at PAD before being sent elsewhere. Now look at screen 125. There's the terminating " , the defining word, MSG:, and the message name, and a short message with ." This final message is just there for convenience in this demo and is not needed in general. Type

```
124 LOAD
```

and when it tells you to put in the destination disk, swap in the extra blank disk you formatted, then press START as directed. At the next prompt, swap back and press START again. When using this method you must be very careful not to reverse your disks or you may wipe out part of your source disk.

Well, the message is now written to the second disk on screen 80, and the word MSGDEM2 knows where to find it. Let's take a quick look to see that it's really there. Type

MTB

to empty the buffers, and then swap disks again (so that the destination disk is in the drive) and do 80 LIST, and then 81 LIST. There's the message. The first two strange bytes on screen 80 are the count. Now do

MTB CR CR MSGDEM2

and watch the routines pull the message from the disk. While we're here, let's send this to the printer. But since your printer may have characteristics different from the printer this package is initialized for, we want to adjust a couple of things. The first item is a quan named PWID. This is the actual number of columns your printer has. The default value is 80. To change it to 96, for example, type

96 TO PWID

The second item is the quan PRTWID which is the width of the area you'd like to print to. The default again is 80. To set it to 60, say, type

60 TO PRTWID

The third item is how far you'd like to indent. This is the quan PRTIND and its initial value is 0. To set it to 10 type

10 TO PRTIND

Finally, we want to tell the formatter to send its output to the printer now, so type

PRT:

(The default setting was to the video display, and will be called back by VID:) Is your printer ready? Let's try it. Type

MSGDEM2

Since many printers will get confused if a character with the high bit set is sent to them you might want to be careful about this.

Incidentally, the same options are available with the video display. PRTWID becomes VIDWID and PRTIND becomes VIDIND. PRT: becomes VID:. There is no "VWID" since the formatter derives this from the positioning of the margins. (The left margin is kept by the OS in the byte at 82 decimal, and the right margin byte is at 83. Default are 2 and 39 respectively.)

Any new printer settings only become active when PRT: is executed, and likewise with video settings and VID:.

Try

```
20 TO VIDWID
4 TO VIDIND
VID:
CR CR MSGDEM2
38 TO VIDWID (back to default)
0 TO VIDIND (ditto)
VID: (move in new values)
```

OK, now swap the source disk back in, that is, the TCAF working disk, but keep the destination disk handy. Let's load a few (six) more messages. Type

```
MTB (to empty the buffers)
126 LOAD
```

and follow the prompts.

As you can see, any large amount of this single-drive compilation could be quite tiresome. Do a short VLIST (abort with any of the three yellow console buttons) and look at the new messages. Swap in the destination disk, do MTB, and then try

```
CR MO
CR M1
etc.
```

Encryption and 2-Drive Systems

There are two more features to point out. They are encryption/decryption (e/d) and adjustments for two-drive systems. Concerning e/d, look at screen 105, or wherever you find the title EN, DECRYPT or similar. (Do an INDEX if you can't find the right screen easily.) Notice that there is a --> at the top of this screen which is causing it not to load. Remove this arrow with your editor. (Since you're going to reload the system in a minute anyway, it's ok if you over-write the system to get an editor in. Get one in somehow.) Now on the next two screens you should find the words ENCRYPT and DECRYPT in parentheses. (DECRYPT is in three times.) Remove the parens to allow these two words to load. Now, you folks with two drives, find the screen where MSG: is defined. (On or near screen 112). There are several sets of parens. Leave the ones that enclose \$ENCRYPT and ... \$DECRYPT ... alone. Shift only the ones that are around " DR! or " to be around " or DSTDSK " and shift the ones around " DRO or " to be around " or SRCDSK ." Just to be on the safe side, here's a picture of how the screens should look after these changes.

FOR ONE OR TWO DRIVE SYSTEMS

FOR TWO DRIVE SYSTEMS ONLY

Scr # 105

```

0 ( Vrtxt: EN,DECRYPT example )
1
2
3
4 : ENCRYPT          ( c1 -- c2 )
5   117 - DUP 0(
6   IF 256 + ENDIF ;
7
8 : DECRYPT          ( c2 -- c1 )
9   117 + DUP 255 )
10  IF 256 - ENDIF ;
11
12
13
14
15      -->

```

Scr # 106

```

0 ( Vrtxt: V$TP V$@ V$*EMT )
1
2 : V$TP          ( -- X$ [=PAD] )
3   VRTC@ DECRYPT NXTVRT VRTC@
4   DECRYPT NXTVRT 256 * + ;
5
6 : V$@          ( -- X$ [=PAD] )
7   PAD 2+ V$TP DUP PAD ! 0
8   DO VRTC@ OVER C! 1+ NXTVRT
9   LOOP DROP PAD ;
10
11 : V$*EMT       ( -- X$ [=PAD] )
12   V$TP 0
13   DO VRTC@ DECRYPT
14   *EMIT NXTVRT
15   LOOP ;      -->

```

Scr # 107

```

0 ( Vrtxt: V$! )
1
2 : V$!          ( X$ -- )
3   DUP @ 2+ 0
4   DO DUP C@ ENCRYPT
5   VRTC! 1+ NXTVRT
6   LOOP DROP ;
7
8
9
10
11
12
13
14
15      -->

```

Scr # 112

```

0 ( Vrtxt: ALT$! MSG: )
1
2 : ALT$!          ( X$ -- )
3   VRTSAV ALTREC V$!
4   ALTSAV VRTREC ;
5
6 : MSG:          ( X$ -- )
7   ( $ENCRYPT )
8   <BUILDS DR1 ( or DSTDSK )
9   ALTBLK , ALTIN , ALT$!
10  FLUSH DR0 ( or SRCDSK )
11  DOES> VRTSAV
12  DUP @ SWAP 2+ @ IN ! BLK !
13  V$*EMT ( or )
14  ( V$@ $DECRYPT XCOUNT *TYPE )
15  VRTREC ;

```

What this last change does is substitute an automatic disk-shift for the swap prompts driven by DSTDSK and SRCDSK. These words are no longer needed and may be bypassed later on, if you're comfortable with the way things are working. OK, now, although some of you might be able to get these changes in by doing some FORGETting and reloading (if you didn't have to overwrite the system before), why not just reload the whole (modified) system this time, starting from valFORTH 1.1. Don't forget the debugger if you want it, and remember to initialize with TYPEOUT. Go ahead. We'll wait.

Now you can repeat the exercises from before, starting where you did the 124 LOAD. Two drive systems should have the "destination" disk, the one with the messages on it, in the second drive, and the source disk, TCAF, in the first drive. Two-drive systems should execute DR1 (which in FORTH means the second drive, which is number 2 on Atari systems) before saying message names so that the code will read the right drive. Say DR0 to go back to the source-code drive. (Released programs will of course not want to say DR1 since they will expect the "game" or application disk to be in the first drive, DR0, which is default. When you are making messages in this way, be sure to start them high enough on disk that your AUTO'd program, which will actually do the message retrieval, will fit under them.) If you look at screen 80 on the destination disk, what you'll see is that the text is now scrambled. The encryption routine used is a simple off-set scheme, and would be easy to crack for a serious hobbyist, though not for the casual user. If you are interested in a higher degree of security, you can encode a whole string at a time with more sophisticated routines. A pair of names, \$ENCRYPT and \$DECRYPT, have been reserved for these routines. We don't provide any examples, but a modern text on cryptography might be a good place to start looking. Anyway, if you use the names \$ENCRYPT and \$DECRYPT, and if the routines expect an extended string on stack (that is, one with a two-byte count at its front end) then they will (hopefully) snap right into the spot designated by the parens.

Windows

Windows are rectangular areas of the video display. They are not supported on the printer, but are supported in both "black and white" graphics 0 mode, and colored graphics 1 and 2 modes. Windows may be set up on-the-fly or they may be given names so that words can call them up readily. The implementation provided here may be used as an example and guide, since you may want your windows to act somewhat differently.

Since it is tricky to interact with a graphics 0 window from the keyboard (there is no simple way that we can find to date to create a split-screen option) we'll illustrate windows in graphics 1, and so also show how color works. Type

```
1 GR.  
3 4 10 5 MAKECW
```

This makes a Color Window whose upper left hand corner is at the 3rd column over, 4th row down (counting the left and top edges as zero), and which is

10 characters wide and 5 high. We have messages M0 through M5 still available, so let's send them to the window. Type

```
WINDOUT (counterpart of TYPEOUT)
OFF CAPS
0 COLOR M0
1 COLOR M1
2 COLOR M2
3 COLOR M3
WCLR
```

Note the extra coloration caused by the mix of upper and lower case. This can be canceled by ON CAPS though it restricts the user to two color for the letters instead of four. A good practice would be to put all the source text for colored windows in upper case. Coloration switches in the middle of a message could be implemented by control characters similar to the right-arrow character and its meaning of *CR. This is done in *EMIT, you'll remember, and you might even use a case statement.

Numerical formatting is also supported, by the words *. and *.R which are direct counterparts of . and .R, except that they go through the formatter before outputting. Try

```
3456 *. *CR
7890 7 *.R *CR
```

These routines should be used both with WINDOUT and TYPEOUT. Using just . or .R will upset the formatting.

There is also

```
2 2 15 8 NAMECW BINGO
```

which names a color window with the given parameters as BINGO. When BINGO is executed it will clear itself and position the imaginary cursor at its the upper lefthand corner. By studying the code, this and other performance characteristics may be altered.

Text Compression

Finally, there is "true" text compression (TC) itself. TC is intended primarily for applications where disk access to messages is not available, such as in cassette-booted systems. This utility uses Transient structures which you have probably come across before in the packages in this series. Hence, all the warnings about memory collisions must to some extent apply. The text compression utilities themselves are fairly straightforward to use, but what they do is rather complex. Briefly, TC allows the creation of bits of headerless code called "tc-texts" that, when executed, put a string onto the stack and then jump to the appropriate Forth words, for formatting. These tc-texts come in three types in this package, namely, tc-words, tc-suffixes and tc-prefixes.

The general procedure is to:

- (1) Load the text compression routines.
- (2) Define all needed tc-texts.
- (3) Define all words that use tc-texts within themselves.
- (4) Execute DISPOSE which will sever links to all of the tc-text creating and compiling structures, leaving only minimal, minimal, headerless structures.

Since (it turns out) we can load the text-compression routines right on top of the rest of the code we already have in, let's do that. Look at the screen in the load chain which you modified at the beginning of this excursion. It was probably 167. Text compression was not loaded at that time. Note the load screen for text compression (probably screen 60) and load it. Because this section uses transients you cannot use SAVE to create a bootable copy until you have executed DISPOSE to break the links to the transient area. In addition, FORGET will act a bit odd, and may cause crashes, so try to avoid using it until you have disposed. There are only three new words to learn in text compression, namely, W= , P= , and S=. These words define tc-words, tc-prefixes and tc-suffixes. For example,

```
W= DOG
W= TAIL
P= SUPER
S= 'S
S= !
```

defines five tc-texts. Type in the five definitions above and then type

```
DOG DOG DOG DOG DOG CR *CR
SUPER DOG CR *CR
SUPER DOG 'S TAIL ! CR *CR
```

The justification, capitalization, coloring, window output, and other options will also function with tc-texts.

Obviously, there is potential for numerous word-name conflicts between tc-texts and FORTH. The punctuation marks, for instance P= . P= , P= ! and so on all are desirable and all already exist in the FORTH vocabulary. Hence the three defining words for tc-texts automatically put the words they define into a separate vocabulary named ^ . In addition, the name | (Shift-=) has been assigned as an alias for FORTH to shorten source code and ease typing. For instance, one might have a FORTH word like:

```
: ?TL ( flag -- )
  IF ^ A_DOG_'S_TAIL_IS_HERE ! *CR |
  ENDIF ;
```

By going into the ^ vocabulary the tc-text ! was interpreted properly, instead of as the FORTH !. Similarly, by going back into the | (FORTH) vocabulary, the word ; was interpreted as the FORTH ; rather than as some prefix that might have been in the ^ vocabulary.

Several more points are worth noting:

* If you are programming short phrases that do not generally run together, you can save some memory by defining a <BUILDS DOES> construct that always attaches the *CR to the end of the operation, thus saving two bytes per message, with the new <BUILD DOES> that loaded with this package.

* If you want to create new types of tc-texts, such as one to deal with problems like SHINE ING, just follow the examples of how the words W= P= S= are constructed. Smart prefixes that strip trailing vowels, for example, would not be difficult to code, but would not necessarily be worth the memory cost. However, in a very large application it might well be worth coding a large number of spelling rules.

* To create tc-texts that contain blanks, create a control character that is not printed, and use this as the blank. The character we suggest for this is the underline, whose ATASCII is 95. Note how the right-arrow, ATASCII 31, is picked off by *EMIT. Do the same for 95, only make it perform *SPACE instead of *CR as right-arrow does.

* Some tc-texts will get rather long, and will be cumbersome in source text. They can be provided with a no-cost alias. For example, say we had

```
W= A_DOG_WITH_A_BONE
```

We could then add

```
TRANSIENT
: D&B ^ [COMPILE] A_DOG_WITH_A_BONE_ | ; IMMEDIATE
PERMANENT
```

This alias would be removed by DISPOSE, as would, of course, the tc-text name A_DOG_WITH_A_BONE, leaving only the headerless tc-text itself.

* As set up, the Transient system is 4000 bytes below the display list. This may not be enough for some applications. The way to find out how much room you have left in the transient area is to type

```
TRANSIENT 741 @ HERE - U. PERMANENT
```

You might even define a word to do this. Call it TFREE. A trap in CREATE, and so also in : is designed to keep you from actually running into the display list by simply aborting the definition in progress when there are less than 128 bytes left.

* Finally, always remember to DISPOSE when you're done with the transients. If you forget and do SAVE you will not get a working system.

This system of compression is quite compact, costing only two bytes to produce output from a tc-text. The cost in memory of producing the tc-text of a word of n letters, (not even counting the trailing blank) is only n + 2.

TEXT COMPRESSION AND AUTO TEXT FORMATTING GLOSSARY

Basic Commands

*." (--)

Like ." , but sends string to the active formatting/outputting routines.

*TYPE (addr count --)

Like TYPE, but sends string of count characters starting at addr to the active formatting/outputting routines.

*CR (--)

Somewhat like CR in that it causes a carriage return. In addition, *CR first formats and flushes the buffer to the output device, and clears the buffer after doing so.

*EMIT (c --)

Like EMIT except sends the character c to the formatter, instead of directly to the output device.

*SPACE (--)

Sends a single character of value in the quan BKGND to the formatter, through *EMIT.

*SPACES (n --)

Sends n characters of value in the quan BKGND to the formatter, through *EMIT.

*BACKS (--)

Similar to action of delete key. Backs up the formatter buffer pointer, BPTR, one location and fills new location with BKGND value.

RGTJST (--)

Sets up formatter for right justification.

LFTJST (--)

Sets up formatter for left justification.

CTRJST (--)

Sets up formatter for center justification.

FILJST (--)

Sets up formatter for fill justification.

INVID (f --)

ON INVID means text will be output in inverse video; OFF INVID means normal video.

INVBK (f --)

ON INVBK means background of text will be output in inverse video. OFF INVID means normal video.

CAP (--)

Causes capitalization of the next byte processed by *EMIT or *TYPE.

CAPS (f --)

ON CAPS means subsequent formatted text will be capitalized if lower case. OFF CAPS means text will be printed as-is.

COLOR (b --)

New color register b will be used for color of subsequent text output to windows in Graphics modes 1 and 2.

TYPEOUT (--)

Initialization routine for the formatter. Either TYPEOUT or WINDOUT must be executed before the first attempt to output text from the formatter or the system may crash. TYPEOUT directs the formatter to use TYPE as its actual output routine, allowing output to the display screen or printer.

WINDOUT (--)

Initialization routine for the formatter. Either TYPEOUT or WINDOUT must be executed before the first attempt to output text from the formatter or the system may crash. WINDOUT directs the formatter to use window routines for output. A window must be created before attempting to use window output or the system may crash. See also NAMWND.

Quans, vects, and subcommands

FDIR (-- +-1)

A quan that holds the next direction to be used by the fill-justification routines when padding the text in the formatting buffer with blanks.

*JUST (--)

A vect used to point to the routine that performs whatever justification action is current. Altered by LFTJST, RGTJST, CTRJST, and FILJST.

BKGND (-- n)

Quan which holds the value of the background character to be used when clearing the formatting buffer. Generally either 32 (blank) or 160 (inverse blank.) See INVBK.

EOB (-- n)

Quan which points to the location in the formatter buffer corresponding to the last allowable position in the current output width. Set up by various routines including PRT:, VID:, and window-creating routines. Stands for "end of buffer."

BPTR (-- n)

Quan which points to the next available location in the formatter buffer. May be user-altered for special purposes, but should not be placed lower than BUF or higher than EOB. Stands for "buffer pointer."

WWID (-- n)

Quan which holds width of field to which text will be output. Used to set up EOB, which is actually used by the formatting routines. See EOB. Stands for "window width" though windows as defined elsewhere need not exist.

*XMTLN (--)

A vect that points to the routine to be used to move text from the formatter buffer to the output device. Set up at present either by TYPEOUT or WINDOUT. Stands for "transmit line."

BUF (--)

A label that points to the beginning of the formatter buffer area. This area need only be three bytes longer than the longest line to be formatted.

INVBK (ON or OFF --)

When ON, background character output by formatter in 0 graphics mode will be inverse video blank. When OFF, this character will be normal video blank. Sets up BKGND. See BKGND.

BUFCLR (--)

Fills the formatter buffer with BKGND.

BUFINIT (--)

Fill the formatter buffer with BKGND, sets up EOB using WWID and BUF, and points sets BPTR equal to BUF.

*TINT (c -- c)

A vect that either points to the coloring routines when a color window is active, or to NOOP when a 0 graphics window is active.

*CAP (c -- c)

Capitalization routine.

*INV (c -- c)

A vect that either points to the inversing routine when a 0 graphics window is active, or to NOOP when a color window is active.

Text Compression

W= xxx, (--)
xxx: (--)

Creates a tc-word-compiling word, named xxx, and a headerless tc-word which when executed sends the string xxx through the formatter followed by *SPACE. xxx when executed, compiles in the cfa of this tc-word. W= and xxx are both in transient area and so are disposed by DISPOSE.

P= xxx, (--)
xxx, (--)

Creates a tc-prefix-compiling word, named xxx, and a headerless tc-prefix which when executed sends the string xxx through the formatter. xxx when executed, compiles in the cfa of this tc-prefix. P= and xxx are both in the transient area and so are disposed by DISPOSE.

S= xxx, (--)
xxx, (--)

Creates a tc-suffix-compiling word, named xxx and a headerless tc-suffix which when executed sends the string xxx through the formatter preceded by *BACKS and followed by *SPACE. xxx, when executed, compiles in the cfa of this tc-suffix. S= and xxx are both in the transient area and so are disposed by DISPOSE.

Typed Output

PRTWID (-- n)

A quan containing the width of the area to be printed when printer output from the formatter has been selected by PRT:. PRT:, among other things, moves PRTWID to WWID.

PRTIND (-- n)

A quan containing the number of spaces the printer is to indent when outputting from the formatter. PRTIND is moved to PVIND by PRT:

PVIND (-- n)

A quan containing the number of spaces the output device is to indent when outputting from the formatter. Set up by PRT: from PRTIND or by VID: from VIDIND.

PWID (-- n)

A quan containing the number of columns the printer is actually able to print as it is currently configured, and independent of the formatting routines.

VIDIND (-- n)

A quan containing the number of spaces the output routines is to indent when outputting from the formatter. VIDIND is moved into PVIND by VID:.

VIDWID (-- n)

A quan containing the width of the area to be written when video output from the formatter has been selected by VID:. VID:, among other things, moves VIDWID to WWID.

PRT: (--)

Directs TYPed output to the printer, and moves appropriate values into WWID and PVIND.

VID: (--)

Directs TYPed output to the video display, and moves appropriate values into WWID and PVIND.

PRINIT (--)

Resets PCTR, the printed line counter.

*XMTLNP (--)

Routine sent to the vect *XMTLN by TYPEOUT. Routes output through TYPE.

Windows

WADR (--)

Address in memory corresponding to character position in upper lefthand corner of current window.

WHGT (--)

Height in lines of currently active window.

LPTR (--)

Counter that holds number of next line in window to which text is to be written. If LPTR points beyond the window then scrolling will occur at next output.

B/LN (-- n)

Bytes per line. Necessary datum for scrolling and clearing routines for windows.

WCLR (--)

Fills the current window with BKGND.

NAMWND (wadr wid hgt b/ch byt/ln --)

One of many possible window-defining structures. Accepts window upper lefthand corner address, its width, height, byte-character, and the bytes/ln of the current graphics mode.

NAMEBW xxx, (column row wid hgt --)
xxx: (--)

Names a 0 graphics window for later activation.

MAKEBW (col row wid hgt --)

Establishes a 0 graphics window immediately but does not name it for later retrieval.

NAMECW xxx, (col row wid hgt --)
xxx: (--)

Names a 1 or 2 graphics window for later activation.

MAKECW (col row wid hgt --)

Establishes a 0 graphics window immediately but does not name it for later retrieval.

Virtual (Disk-based) Memory

(A pointer to a byte on disk is implemented by the two system variables, BLK and IN in the fig model. BLK contains the block number pointed to and IN contains the number of bytes into the block the byte in question is located.)

VRTC@ (-- b)

Fetches the byte pointed to on disk by the system variable BLK and IN. (BLK is the block number, and IN is the number of bytes into the block the desired byte is located.)

VRTC! (b --)

Stores the byte on stack to the location on disk pointed to by BLK and IN. See VRTC@.

VRTSAV (--)

Saves the values of system variables BLK and IN to quans OBLK and OIN respectively.

VRTREC (--)

Recalls the values of the system variables BLK and IN from the quans OBLK and OIN respectively.

NXTVRT (--)

Bumps the system variables BLK and IN as required to point to the next location in virtual memory.

RELVRT (offset --)

Takes an offset on stack and alters the system variables BLK and IN as necessary to point offset bytes from their initial virtual memory location.

V" (-- blk in)

Leaves the values of BLK and IN on the stack at the time it is executed and then scans the virtual memory pointer formed by BLK and IN forward until the next " character is encountered.

XMTV (--)

Starting from the location in virtual memory pointed to by BLK and IN, outputs characters through *EMIT until a " character is encountered, which it does not output.

XCOUNT (adr -- adr+2 xcount)

Extracts a two-byte count from an extended string, and leaves the count on top of the address + 2.

M: xxx, (blk in --)
xxx: (--)

Generally used after V". Takes a virtual memory pointer from the stack, and creates a word xxx which when executed will push the virtual memory pointer to BLK and IN and then execute XMTV, thus retrieving a message from disk. See Strolling... for an example.

V: xxx, (blk in --)
xxx: (--)

Creates a word xxx which when executed pushes the virtual memory pointer which was on stack at the time of its creation to BLK and IN.

V\$TP (-- XCOUNT)

Extracts a two-byte string count from the disk location to which BLK and IN point, leaves it on stack, and bumps the virtual memory pointer made up of BLK and IN twice.

V\$@ (-- X\$=PAD)

Extracts the extended string in virtual memory pointed to by BLK and IN. The string is left at PAD.

V\$*EMT (--)

Sends the extended string pointed to by BLK and IN through *EMIT.

V\$! (X\$ --)

Stores the extended string on stack to virtual memory starting at the location pointed to by BLK and IN.

X" (---X\$=PAD)

Reads the following characters until the delimiter " as an extended string and stores the string at PAD. Operates from screens only. Crosses block and screen boundaries without additional code. Do not use --> to cross screens, as --> will just become part of the string.

ALTSV (--)

Copies variables BLK and IN to quans ALTBK and ALTIN respectively.

ALTREC (--)

Copies quans ALTBK and ALTIN to variable BLK and IN respectively.

ALTINIT. (scr --)

Sets up ALTBK and ALTIN to point to screen scr. ALTBK and ALTIN form an auxiliary virtual memory pointer that is used to keep track of how far messages have been compiled onto the destination disk.

ALT\$! (X\$ --)

Like V\$! except stores string through alternate virtual memory pointers made up of ALTBK and ALTIN.

LI. TEXT COMPRESSION AND AUTO TEXT FORMATTING
SUPPLIED SOURCE LISTING

Screen: 1

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 4

0 (Transients: setup)
1 ' (QUAN) (5 KLOAD)
2
3 BASE @ DCX
4
5 HERE
6
7 741 @ 4000 - DP !
8 (SUGGESTED PLACEMENT OF TAREA)
9
10
11 HERE CONSTANT TAREA
12 QUAN TP
13 QUAN TPFLAG 1 TO TPFLAG
14 QUAN OLDDP (old HERE) TO OLDDP
15 -->

Screen: 2

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 5

0 (Xsients: TRANSIENT PERMANENT)
1 (Expanded from code by Phillip)
2 (Wasson, in Forth Dimensions)
3
4 : TRANSIENT (--)
5 TPFLAG NOT
6 IF HERE TO OLDDP TP DP !
7 1 TO TPFLAG
8 ENDIF ;
9
10 : PERMANENT (--)
11 TPFLAG
12 IF HERE TO TP OLDDP DP !
13 0 TO TPFLAG
14 ENDIF ;
15 -->

Screen: 3

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 6

0 (Transients: DISPOSE)
1 : DISPOSE PERMANENT
2 CR ." Disposing..." VOC-LINK
3 BEGIN DUP 0 53279 C!
4 BEGIN @ DUP TAREA U(
5 UNTIL DUP ROT ! DUP 0=
6 UNTIL DROP VOC-LINK @
7 BEGIN DUP 4 -
8 BEGIN DUP 0 53279 C!
9 BEGIN PFA LFA @ DUP TAREA U(
10 UNTIL
11 DUP ROT PFA LFA ! DUP 0=
12 UNTIL DROP @ DUP 0=
13 UNTIL DROP [COMPILE] FORTH
14 DEFINITIONS ." Done" CR ;
15 PERMANENT BASE !

Screen: 7

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 10

0 (Quan: ASSIGN)
1
2
3
4 ' (CFALIT
5 : ASSIGN [COMPILE] CFALIT ;
6 IMMEDIATE -->)()
7
8 : ASSIGN (-- cfa)
9 STATE @
10 [COMPILE] [
11 [COMPILE] ' CFA SWAP
12 IF]
13 ENDIF [COMPILE] LITERAL ;
14 IMMEDIATE
15 -->

Screen: 8

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 11

0 (Quan: TO AT)
1
2 : TO
3 -FIND @= @ ?ERROR DROP
4 STATE @
5 IF ,
6 ELSE EXECUTE
7 ENDIF ; IMMEDIATE
8
9 : AT
10 -FIND @= @ ?ERROR DROP
11 2+ STATE @
12 IF ,
13 ELSE EXECUTE
14 ENDIF ; IMMEDIATE
15 (corrected) -->

Screen: 9

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 12

0 (Quan: [2@6] [2!4])
1
2 ASSEMBLER HEX
3
4 LABEL (2@6)
5 A0 C, 06 C, B1 C, W C, 48 C,
6 C8 C, B1 C, W C, 4C C, PUSH ,
7
8 LABEL (2!4)
9 A0 C, 04 C, B5 C, 00 C, 91 C,
10 W C, C8 C, B5 C, 01 C, 91 C,
11 W C, 4C C, POP ,
12
13
14
15 -->

Screen: 13

```

0 ( Quan: [2V6] )
1
2 LABEL (2V6)
3 A0 C, 07 C, B1 C, W C, 48 C,
4 88 C, B1 C, W C, 85 C, W C,
5 68 C, 85 C, W 1+ C,
6 A0 C, 00 C, 4C C, W 1- ,
7
8
9
10
11
12
13
14
15 -->

```

Screen: 16

```

0 ( Utils: UMAX UMIN HIDCHR )
1
2 : UMAX ( u1 u2 -- u3 )
3 2DUP U(
4 IF SWAP ENDIF
5 DROP ;
6
7 : UMIN ( u1 u2 -- u3 )
8 2DUP U)
9 IF SWAP ENDIF
10 DROP ;
11
12 '( HIDCHR )(
13 : HIDCHR
14 -1 94 ! ; )
15 -->

```

Screen: 14

```

0 ( Quan: patch for CREATE )
1
2 DCX
3
4 : (PTCH) ( system )
5 SWAP >R R = 251 R = 249 R) =
6 OR OR ;
7
8 : PTCH ( system )
9 IF [ ' (PTCH) CFA ] LITERAL
10 ELSE [ ' = CFA ] LITERAL
11 ENDIF
12 [ ' CREATE 63 + ] LITERAL ! ;
13
14
15 -->

```

Screen: 17

```

0 ( Utils: S: P: )
1
2 '( S: ;S )( )
3 HEX
4
5 : S: ( f -- )
6 PFLAG @ SWAP
7 IF 1 OR ELSE FE AND ENDIF
8 PFLAG ! ;
9
10 : P: ( f -- )
11 PFLAG @ SWAP
12 IF 2 OR ELSE FD AND ENDIF
13 PFLAG ! ;
14
15 DCX

```

Screen: 15

```

0 ( Quan: QUAN VECT )
1
2 : QUAN
3 ON PTCH LABEL -2 ALLOT
4 (206) , (2!4) ,
5 [ ' VARIABLE 4 + ] LITERAL ,
6 2 ALLOT OFF PTCH ;
7
8 : VECT
9 ON PTCH LABEL -2 ALLOT
10 (2V6) , (2!4) ,
11 [ ' VARIABLE 4 + ] LITERAL ,
12 [ ' NOOP CFA ] LITERAL ,
13 OFF PTCH ;
14
15

```

Screen: 18

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 19

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 22

0 (Screen code conversion words)
1
2 SWAP ! 91 C, C4 C, 68 C, 29 C,
3 80 C, 11 C, C4 C, 91 C, C4 C,
4 C8 C, D0 C, D3 C, E6 C, C7 C,
5 E6 C, C5 C, 4C C, ,
6
7
8 : >SCD SP@ DUP 1 >BSCD ;
9 : SCD> SP@ DUP 1 BSCD> ;
10
11
12
13
14
15
BASE !

Screen: 20

0 (Screen code conversion words)
1
2 BASE @ HEX
3
4 CODE >BSCD (a a n --)
5 A9 C, 03 C, 20 C, SETUP ,
6 HERE C4 C, C2 C, D0 C, 07 C,
7 C6 C, C3 C, 10 C, 03 C, 4C C,
8 NEXT , B1 C, C6 C, 48 C,
9 29 C, 7F C, C9 C, 60 C, B0 C,
10 0D C, C9 C, 20 C, B0 C, 06 C,
11 18 C, 69 C, 40 C, 4C C, HERE
12 2 ALLOT 38 C, E9 C, 20 C, HERE
13 SWAP ! 91 C, C4 C, 68 C, 29 C,
14
15 -->

Screen: 23

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 21

0 (Screen code conversion words)
1
2 80 C, 11 C, C4 C, 91 C, C4 C,
3 C8 C, D0 C, D3 C, E6 C, C7 C,
4 E6 C, C5 C, 4C C, , C;
5
6 CODE BSCD> (a a n --)
7 A9 C, 03 C, 20 C, SETUP ,
8 HERE C4 C, C2 C, D0 C, 07 C,
9 C6 C, C3 C, 10 C, 03 C, 4C C,
10 NEXT , B1 C, C6 C, 48 C,
11 29 C, 7F C, C9 C, 60 C, B0 C,
12 0D C, C9 C, 40 C, B0 C, 06 C,
13 18 C, 69 C, 20 C, 4C C, HERE
14 2 ALLOT 38 C, E9 C, 40 C, HERE
15 -->

Screen: 24

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 25

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 28

0 (AF0: quans vects)
1
2 QUAN BKGND (background chr)
3 BL TO BKGND
4 QUAN EOB (end of buffer)
5 QUAN BPTR (buffer pointer)
6 QUAN WWID (characters/line)
7 QUAN B/C (bytes/character)
8 QUAN LWD (1st chr of last wd)
9 VECT *XMTLN (send fmed ln)
10
11 LABEL BUF 123 ALLOT (buffer)
12 (Need only be longest line +3)
13
14
15 -->

Screen: 26

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 29

0 (AF0: ?BL INVBK)
1
2 : ?BL (-- f)
3 C@ 31 AND @= ;
4
5 : INVBK (f --)
6 IF 160
7 ELSE BL
8 ENDIF TO BKGND ;
9
10
11
12
13
14
15 -->

Screen: 27

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 30

0 (AF0: BUFCLR BUFINIT)
1
2 : BUFCLR (--)
3 BUF WWID BKGND FILL ;
4
5 : BUFINIT (--)
6 WWID BUF + 1- TO EOB
7 BUFCLR BUF TO BPTR ;
8
9
10 38 TO WWID
11 BUFINIT
12
13 (Setup for 0 GR. display)
14
15

Screen: 31

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 34

0 (R,cjust: ^RJ ^CJ)
1
2 ' (*JUST) (16 KLOAD)
3 : (RCJ) (b --)
4 >R BUF BUF EOB LCHR -
5 R) / DUP >R +
6 LCHR BUF - 1+ <MOVE
7 BUF R) BKGND FILL ;
8
9 : ^RJ (--)
10 1 (RCJ) ;
11
12 : ^CJ (--)
13 2 (RCJ) ;
14
15 -->

Screen: 32

0 (Justify: *JUST LCHR ^LCHR)
1
2 VECT *JUST
3 QUAN LCHR
4
5 : ^LCHR (--)
6 EOB
7 BEGIN DUP BUF U) OVER ?BL AND
8 WHILE 1-
9 REPEAT TO LCHR ;
10
11
12
13
14
15

Screen: 35

0 (R,cjust: RGT,LFT,CTRJST)
1
2 : RGTJST (--)
3 ASSIGN ^RJ TO *JUST ;
4
5 : LFTJST (--)
6 ASSIGN NOOP TO *JUST ;
7
8 : CTRJST (--)
9 ASSIGN ^CJ TO *JUST ;
10
11
12
13
14
15

Screen: 33

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 36

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 37

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 40

0 (Fjust: ^FJ FILJST)
1
2 : ^FJ (--)
3 1 TO ?FJ
4 BEGIN LCHR EOB U(?FJ AND
5 WHILE FDIR 0)
6 IF BUF ELSE LCHR ENDIF
7 TO FPTR FPASS
8 REPEAT FDIR MINUS TO FDIR ;
9
10 : FILJST (--)
11 ASSIGN ^FJ TO *JUST ;
12
13
14
15

Screen: 38

0 (Fjust: quans (FPTR))
1
2 ' (*JUST) (16 KLOAD)
3
4 QUAN FDIR 1 TO FDIR
5 QUAN FPTR
6 QUAN ?FJ
7
8 : (FPTR) (f --)
9 FPTR BUF U(NOT
10 FPTR LCHR U) NOT AND ;
11
12
13
14
15 -->

Screen: 41

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 39

0 (Fjust: FPASS)
1
2 : FPASS (--)
3 0 TO ?FJ
4 BEGIN LCHR EOB U((FPTR) AND
5 WHILE FPTR ?BL
6 IF 1 TO ?FJ
7 FPTR FPTR 1+ EOB FPTR -
8 (CMOVE 1 AT LCHR +!
9 BEGIN FDIR AT FPTR +!
10 FPTR ?BL NOT (FPTR) NOT OR
11 UNTIL
12 ENDIF FDIR AT FPTR +!
13 REPEAT ;
14
15 -->

Screen: 42

0 (AF1: [LWD] MOVWD RETWD)
1 ' (BKGND) (;S)
2
3 : (LWD) (--)
4 BPTR
5 BEGIN 1- DUP BUF U(
6 OVER ?BL OR
7 UNTIL 1+ TO LWD ;
8
9 : MOVWD (--)
10 LWD HERE BPTR LWD - (CMOVE ;
11
12 : RETWD (--)
13 HERE BUF BPTR LWD -
14 DUP >R CMOVE
15 R) BUF + TO BPTR ; -->

Screen: 43

```

0 ( AF1: [SNDLN] SENDLN )
1
2 : (SNDLN) ( -- )
3 ' ( *JUST ^LCHR *JUST ) ( )
4 *XMTLN BUFCLR ;
5
6 : SENDLN ( -- )
7 (LWD) LWD BUF U)
8 IF MOVWD LWD EOB LWD - 1+
9 0 MAX BKGND FILL
10 ENDIF (SNDLN)
11 LWD BUF )
12 IF RETWD
13 ELSE BPTR 1- C@ BUF C!
14 BUF 1+ TO BPTR
15 ENDIF ; -->

```

Screen: 44

```

0 ( AF1: *CR )
1
2 : *CR ( -- )
3 BPTR BUF =
4 IF BUF WWID BKGND FILL
5 ' ( *JUST
6 ELSE ^LCHR ) ( )
7 ' ( ^FJ ASSIGN ^FJ ) ( 0 )
8 ' ( *JUST
9 AT *JUST @ < )
10 IF *JUST
11 ENDIF ( )
12 ENDIF
13 *XMTLN BUFINIT ;
14
15 -->

```

Screen: 45

```

0 ( AF1: *EMIT )
1
2 : *EMIT ( c -- )
3 DUP 31 =
4 IF DROP *CR
5 ELSE ' ( *TINT *TINT ) ( )
6 ' ( *CAP *CAP ) ( )
7 ' ( *INV *INV ) ( )
8 BPTR C! 1 AT BPTR +!
9 BPTR EOB 1+ U)
10 IF BPTR 1- ?BL
11 IF BPTR EOB 2+ MIN TO BPTR
12 ELSE SENDLN
13 ENDIF
14 ENDIF
15 ENDIF ; -->

```

Screen: 46

```

0 ( AF1: *TYPE )
1
2 : *TYPE ( addr count -- )
3 BEGIN DUP 0)
4 WHILE
5 OVER C@ 127 AND
6 *EMIT 1- SWAP 1+ SWAP
7 REPEAT 2DROP ;
8
9
10
11
12
13
14
15 -->

```

Screen: 47

```

0 ( AF1: *SPACE[S] *BACKS )
1
2 : *SPACE ( -- )
3 BKGND *EMIT ;
4
5 : *SPACES ( n -- )
6 0 MAX -DUP
7 IF 0 DO *SPACE LOOP
8 ENDIF ;
9
10 : *BACKS
11 BPTR 1- BUF UMAX TO BPTR
12 BL ' ( *INV *INV ) ( )
13 BPTR C! ;
14
15 -->

```

Screen: 48

```

0 ( AF1: [*."] *. " )
1
2 : (*.") ( -- )
3 R COUNT DUP 1+
4 R) + >R *TYPE ;
5
6 : *. "
7 ASSIGN TYPE ASSIGN (." )
8 [ ' ." 13 + ] LITERAL
9 ASSIGN (*.") OVER !
10 [ ' ." 35 + ] LITERAL
11 ASSIGN *TYPE OVER !
12 [COMPILE] ."
13 <ROT ! ! ; IMMEDIATE
14
15

```

Screen: 49

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 52

0 (Capitalization: CAP[S] etc.)
1
2 QUAN ?CAP
3 QUAN ?CAPLK
4
5
6 : CAP (--)
7 1 TO ?CAP ;
8
9 : CAPS (f --)
10 DUP TO ?CAPLK TO ?CAP ;
11
12 OFF CAPS
13
14
15 -->

Screen: 50

0 (Coloring: *TINT etc.)
1
2 '(>SCD)(10 KLOAD)
3
4 VECT *TINT
5
6 '(CLRBYT)(
7 0 VARIABLE CLRBYT
8 : COLOR CLRBYT ! ;)
9
10 : ^TINT (c -- c)
11 >SCD CLRBYT @
12 64 * OR SCD ;
13
14 ASSIGN ^TINT TO *TINT
15

Screen: 53

0 (Capitalization: *CAP)
1
2 : *CAP (c -- c)
3 ?CAP
4 IF
5 DUP 127 AND DUP
6 122 (= SWAP
7 97)= AND
8 IF 32 -
9 ENDIF ?CAPLK TO ?CAP
10 ENDIF ;
11
12
13
14
15

Screen: 51

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 54

0 (Inverse Video: *INV etc.)
1
2 QUAN ?INV
3 VECT *INV
4
5 : INVID (f --)
6 128 * TO ?INV ;
7
8 : ^INV (c -- c)
9 ?INV OR ;
10
11 ASSIGN ^INV TO *INV
12
13 OFF INVID
14
15

Screen: 55

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 58

0 (Efficient (BUILDS...DOES))
1
2 : DOES
3 COMPILE (;CODE)
4 4C C, (DOES) , ; IMMEDIATE
5
6 : (BUILDS
7 CREATE SMUDGE ;
8
9 DCX
10
11
12
13
14
15

Screen: 56

0 (Efficient (BUILDS...DOES))
1 (Partly after G. B. Lyons)
2 --> (Pick up C, code nxt scr)
3 ASSEMBLER HEX
4
5 LABEL (WIP)
6 W)Y LDA, CLC, 3 # ADC,
7 IP STA, INY, W)Y LDA,
8 0 # ADC, IP 1+ STA,
9 DEY, RTS,
10
11 LABEL (DOES)
12 IP 1+ LDA, PHA, IP LDA, PHA,
13 (WIP) JSR, ' VARIABLE 4 +
14 JMP,
15 DCX -->

Screen: 59

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 57

0 (Efficient (BUILDS...DOES))
1
2 ASSEMBLER HEX
3
4 LABEL ^WIP
5 B1 C, W C, 18 C, 69 C, 03 C,
6 85 C, IP C, C8 C, B1 C, W C,
7 69 C, 00 C, 85 C, IP 1+ C,
8 88 C, 60 C,
9
10 LABEL (DOES)
11 A5 C, IP 1+ C, 48 C,
12 A5 C, IP C, 48 C, 20 C,
13 ^WIP, 4C C, ' VARIABLE 4 + ,
14
15 -->

Screen: 60

0 (Txt comp: TLABEL)
1
2 '(TRANSIENT)(2 KLOAD)
3 '(^WIP)(28 KLOAD)
4
5 TRANSIENT
6
7 : TLABEL (--)
8 HERE TRANSIENT
9 CONSTANT PERMANENT
10 [COMPILE] ASSEMBLER ;
11
12 : I [COMPILE] FORTH ; IMMEDIATE
13 VOCABULARY ^ IMMEDIATE
14
15 PERMANENT -->

Screen: 61

```

0 ( Txt comp: DMCP$ )
1
2 HEX
3
4 TLABEL DCMP$
5   A5 C, IP 1+ C, 48 C,
6   A5 C, IP C, 48 C, 20 C, ^WIP ,
7   CA C, CA C, 18 C, A5 C, W C,
8   69 C, 02 C, 95 C, 00 C,
9   A5 C, W 1+ C, 69 C, 00 C,
10  95 C, 01 C, A0 C, 01 C,
11  C8 C, B1 C, W C,
12  10 C, FB C,
13  88 C, 98 C,
14  A0 C, 00 C, 4C C, PUSH0A ,
15                                -->

```

Screen: 64

```

0 ( Txt comp: W= P= S= )
1
2 : W= ( -- )
3   (W=) , CURRENT @
4   TC= CURRENT ! ;
5
6 : P= ( -- )
7   (P=) , CURRENT @
8   TC= CURRENT ! ;
9
10 : S= ( -- )
11   (S=) , CURRENT @
12   TC= CURRENT ! ;
13
14 PERMANENT
15

```

Screen: 62

```

0 ( Txt comp: [W=] [P=] [S=] )
1
2 TLABEL (W=) ASSEMBLER
3   4C C, DCMP$ ,
4   ] *TYPE *SPACE ;S [
5
6 TLABEL (P=) ASSEMBLER
7   4C C, DCMP$ ,
8   ] *TYPE ;S [
9
10 TLABEL (S=) ASSEMBLER
11   4C C, DCMP$ ,
12   ] *BACKS *TYPE *SPACE ;S [
13
14 DCX
15                                -->

```

Screen: 65

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 63

```

0 ( Txt comp: TC= )
1
2 TRANSIENT
3
4 : TC=
5   [COMPILE] ^ DEFINITIONS
6   HERE >R TRANSIENT
7   <BUILDS [COMPILE] IMMEDIATE
8   LATEST C@ 31 AND >R
9   LATEST 1+ I' R CMOVE
10  R I' + DUP C@ 128 AND SWAP C!
11  R) R) 2- , PERMANENT ALLOT
12  DOES) @ STATE @
13  IF , ELSE EXECUTE ENDIF ; -->
14
15

```

Screen: 66

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

Screen: 67

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 70

0 (Typed out: quans)
1
2 QUAN PRTWID (printer ch/ln)
3 80 TO PRTWID (init value)
4
5 QUAN VIDWID (video ch/ln)
6 38 TO VIDWID (init value)
7
8 QUAN PCTR (printer line ctr)
9 0 TO PCTR (init value)
10
11 QUAN VIDIND (video indent)
12 0 TO VIDIND (init value)
13 QUAN PRTIND (printer indent)
14 0 TO PRTIND (init value)
15 QUAN PVIND (indentation) -->

Screen: 68

0 (Numerics: FMT#)
1
2 : FMT# (f --)
3 IF
4 ASSIGN *TYPE
5 ASSIGN *SPACES
6 ASSIGN *SPACE
7 ELSE
8 ASSIGN TYPE
9 ASSIGN SPACES
10 ASSIGN SPACE
11 ENDIF
12 [' D. 4 +] LITERAL !
13 [' D.R 22 +] LITERAL !
14 [' D.R 24 +] LITERAL ! ;
15 -->

Screen: 71

0 (Typed out: ?CR PWID ?P,VCR)
1
2 VECT ?CR
3 QUAN PFLG (value for PFLAG)
4 QUAN PWID
5 80 TO PWID (adjust to suit)
6
7 : ?PCR (--)
8 WWID PVIND + PWID <=
9 IF CR ENDIF ;
10
11 : ?VCR (--)
12 WWID PVIND +
13 83 C@ 82 C@ - 1+ (
14 IF CR ENDIF ;
15 -->

Screen: 69

0 (Numerics: *. *.R)
1
2 : *. (n --)
3 ON FMT# . OFF FMT# ;
4
5 : *.R (n r --)
6 ON FMT# .R OFF FMT# ;
7
8
9
10
11
12
13
14
15

Screen: 72

0 (Typed out: PRT: PRINIT VID:)
1
2 : PRT: (--)
3 PRTIND TO PVIND
4 PRTWID TO WWID
5 ASSIGN ?PCR TO ?CR
6 2 TO PFLG BUFINIT ;
7
8 : PRINIT (--)
9 0 TO PCTR ;
10
11 : VID: (--)
12 VIDIND TO PVIND
13 VIDWID TO WWID
14 ASSIGN ?VCR TO ?CR
15 1 TO PFLG BUFINIT ; VID: -->

Screen: 73

```
0 ( Typed out: *XMTLNP )
1
2 : *XMTLNP ( -- )
3 PFLAG @
4 PFLG PFLAG !
5 BUF WWID PVIND SPACES TYPE
6 ?CR PFLG 2 =
7 IF 1 AT PCTR +!
8 PCTR 60 = ( lines/page )
9 IF CR CR CR CR CR CR
10 PRINIT
11 ENDIF
12 ENDIF
13 PFLAG ! ;
14
15 -->
```

Screen: 76

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 74

```
0 ( Typed out: TYPEOUT )
1
2 : TYPEOUT ( -- )
3 ASSIGN *XMTLNP TO *XMTLN ;
4
5 ( for buffer fmtng, no windows)
6
7 TYPEOUT
8
9
10
11
12
13
14
15 -->
```

Screen: 77

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 75

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 78

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 79

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 82

0 (Windows: WCLR)
1
2 : WCLR (f --)
3 B/C WHGT * B/LN * WADR + WADR
4 DO I WWID
5 BKGND
6 >SCD FILL
7 B/LN /LOOP
8 0 TO LPTR ;
9
10
11
12
13
14
15 -->

Screen: 80

0 (Windows: quans etc.)
1
2 ' (>SCD) (10 KLOAD)
3
4 QUAN WADR (window uplftr adr)
5 88 @ 2+ TO WADR (crnt. uplft)
6
7 QUAN WHGT (# lines in window)
8 24 TO WHGT (setup for 0 GR.)
9
10 QUAN LPTR (wndw line pointer)
11 0 TO LPTR (default to top)
12
13 QUAN B/LN (bytes/line)
14 40 TO B/LN (setup for 0 GR.)
15 -->

Screen: 83

0 (Windows: NAMWND WSTP RECWND)
1
2 : NAMWND (wadr wid hgt b/ch)
3 <BUILDS (byt/ln --)
4 , , , , ;
5
6 : WSTP (wa wid hgt b/c b/l --)
7 TO B/LN TO B/C TO WHGT
8 TO WWID TO WADR
9 BUF WWID + 1- TO EOB
10 WCLR BUFINIT ;
11
12 : RECWND (system)
13 >R -2 8
14 DO J I + @
15 -2 +LOOP R) DROP WSTP ; -->

Screen: 81

0 (Windows: [SCROLL] SCROLL)
1
2 : (SCROLL) (--)
3 WWID B/C * >R (# to cmove)
4 B/LN B/C * >R (# to advance)
5 R WHGT 1- * WADR + WADR
6 DO I J + I 4 RPICK CMOVE
7 J /LOOP
8 WADR WHGT 1- R) * + R)
9 BKGND >SCD FILL ;
10
11 : ?SCROLL (--)
12 LPTR WHGT =
13 IF (SCROLL) -1 AT LPTR +!
14 ENDIF ;
15 -->

Screen: 84

0 (Windows: *XMTLNW WINDOUT)
1
2 : *XMTLNW (--)
3 ?SCROLL
4 BUF LPTR B/LN * WADR +
5 WWID >BSCD 1 AT LPTR +! ;
6
7 : WINDOUT (--)
8 ASSIGN *XMTLNW TO *XMTLN ;
9
10 WINDOUT
11
12
13
14
15 -->

Screen: 85

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 88

0 (Color windows: CRPM CVCT)
1 '(WADR)(40 KLOAD)
2
3 : CRPM (col row wid hght --)
4 (wa wid hgt b/c b/l)
5 ROT 20 * 4 ROLL +
6 88 @ + (ROT (set up wadr)
7 1 20 ; (b/chr b/ln)
8
9 : CVCT (--)
10 '(*TINT ASSIGN ^TINT
11 TO *TINT)()
12 '((INV ASSIGN NOOP
13 TO *INV)() ;
14
15 -->

Screen: 86

0 (B&W windows: BWPRM BWVCT)
1 '(WADR)(40 KLOAD)
2
3 : BWPRM (col row wid hght --)
4 (wa wid hgt b/c b/l)
5 ROT 40 * 4 ROLL +
6 88 @ + (ROT (set up wadr)
7 1 40 ; (b/chr b/ln)
8
9 : BWVCT (--)
10 '(*TINT ASSIGN NOOP
11 TO *TINT)()
12 '(*INV ASSIGN ^INV
13 TO *INV)() ;
14
15 -->

Screen: 89

0 (Color windows: NAME,MAKECW)
1
2 : NAMECW (col row wid hght --)
3 CPM NAMWND
4 DOES> RECWND CVCT ;
5
6 : MAKECW (col row wid hght --)
7 CPM WSTP CVCT ;
8
9
10
11
12
13
14
15

Screen: 87

0 (B&W windows: NAMEBW MAKEBW)
1
2 : NAMEBW (col row wid hght --)
3 BWPRM NAMWND
4 DOES> RECWND BWVCT ;
5
6 : MAKEBW (col row wid hght --)
7 BWPRM WSTP BWVCT ;
8
9
10
11
12
13
14
15

Screen: 90

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 91

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 94

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 92

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 95

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 93

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 96

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 97

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 100

0 (Vrtxt: VRTADJ VRTCX)
1
2 : VRTADJ (--)
3 ?LOADING
4 IN @ B/BUF >=
5 IF 0 IN ! 1 BLK +!
6 ENDIF ;
7
8 : VRTCX (-- adr)
9 VRTADJ
10 BLK @ BLOCK IN @ + ;
11
12
13
14
15 -->

Screen: 98

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 101

0 (Vrtxt: VRTC@, ! VRTSAV, REC)
1
2 : VRTC@ (-- b)
3 VRTCX C@ ;
4
5 : VRTC! (b --)
6 VRTCX C! UPDATE ;
7
8 QUAN OBLK QUAN OIN
9
10 : VRTSAV (-- blk in)
11 BLK @ TO OBLK IN @ TO OIN ;
12
13 : VRTREC (blk in --)
14 OIN IN ! OBLK BLK ! ;
15 -->

Screen: 99

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 102

0 (Vrtxt: NXTVRT RELVRT)
1
2 : NXTVRT (--)
3 1 IN +! VRTADJ ;
4
5 : RELVRT (offset --)
6 ?LOADING
7 IN @ + B/BUF /MOD BLK +!
8 DUP 0<
9 IF B/BUF + -1 BLK +!
10 ENDIF IN ! ;
11
12
13
14
15 -->

Screen: 103

```

0 ( Vrtxt: V" XMTV XCOUNT )
1
2 : V" ( -- blk in )
3 VRTADJ BLK @ IN @
4 BEGIN VRTC@ 34 = NXTVRT
5 UNTIL ;
6
7 : XMTV ( -- )
8 BEGIN VRTC@ DUP 34 <
9 WHILE *EMIT NXTVRT
10 REPEAT NXTVRT DROP ;
11
12 : XCOUNT ( adr -- adr+2 cnt )
13 DUP @ SWAP 2+ SWAP ;
14
15 -->

```

Screen: 104

```

0 ( Vrtxt: M: V: )
1
2 : M: ( blk in -- )
3 <BUILDS , ,
4 DOES> VRTSAV
5 DUP @ IN !
6 2+ @ BLK !
7 XMTV *CR VRTREC ;
8
9 : V: ( blk in -- )
10 <BUILDS , ,
11 DOES>
12 DUP @ IN !
13 2+ @ BLK ! ;
14
15 -->

```

Screen: 105

```

0 ( Vrtxt: EN,DECRYPT example )
1
2 -->
3
4 : ENCRYPT ( c1 -- c2 )
5 117 - DUP 0<
6 IF 256 + ENDIF ;
7
8 : DECRYPT ( c2 -- c1 )
9 117 + DUP 255 >
10 IF 256 - ENDIF ;
11
12
13
14
15 -->

```

Screen: 106

```

0 ( Vrtxt: V$TP V$@ V$*EMT )
1
2 : V$TP ( -- XCOUNT )
3 VRTC@ ( DECRYPT ) NXTVRT VRTC@
4 ( DECRYPT ) NXTVRT 256 * + ;
5
6 : V$@ ( -- X$ [=PAD] )
7 PAD 2+ V$TP DUP PAD ! 0
8 DO VRTC@ OVER C! 1+ NXTVRT
9 LOOP DROP PAD ;
10
11 : V$*EMT ( -- )
12 V$TP 0
13 DO VRTC@ ( DECRYPT )
14 *EMIT NXTVRT
15 LOOP ; -->

```

Screen: 107

```

0 ( Vrtxt: V$! )
1
2 : V$! ( X$ -- )
3 DUP @ 2+ 0
4 DO DUP C@ ( ENCRYPT )
5 VRTC! 1+ NXTVRT
6 LOOP DROP ;
7
8
9
10
11
12
13
14
15 -->

```

Screen: 108

```

0 ( Vrtxt: X" )
1
2 : X" ( -- X$ [=PAD] )
3 0 PAD ! PAD 2+
4 BEGIN VRTC@ DUP 34 <
5 WHILE OVER C! 1+
6 1 PAD +! NXTVRT
7 REPEAT NXTVRT 2DROP PAD ;
8
9
10
11
12
13
14
15 -->

```

Screen: 109

```
0 ( Vrtxt: )
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15 -->
```

Screen: 110

```
0 ( Vrtxt : ALTSAV, REC )
1
2 QUAN ALTBK QUAN ALTIN
3
4 : ALTSAV ( -- )
5 BLK @ TO ALTBK
6 IN @ TO ALTIN ;
7
8 : ALTREC ( -- )
9 ALTBK BLK !
10 ALTIN IN ! ;
11
12
13
14
15 -->
```

Screen: 111

```
0 ( Vrtxt: ALTINIT SRCDSK DSTDSK )
1
2 : ALTINIT ( screen -- )
3 B/SCR * TO ALTBK
4 @ TO ALTIN ;
5
6
7 : SRCDSK ( -- )
8 CR ." Insert source disk and p
9 res START." WAIT CR ;
10
11 : DSTDSK ( -- )
12 CR ." Insert dest. disk and pr
13 ess START." WAIT CR ;
14
15 -->
```

Screen: 112

```
0 ( Vrtxt: ALT$! MSG: )
1
2 : ALT$! ( X$ -- )
3 VRTSAV ALTREC V$!
4 ALTSAV VRTREC ;
5
6 : MSG: ( X$ -- )
7 ( $ENCRYPT )
8 (BUILDS ( DR1 or ) DSTDSK
9 ALTBK , ALTIN , ALT$!
10 FLUSH ( DR0 or ) SRCDSK
11 DOES) VRTSAV
12 DUP @ SWAP 2+ @ IN ! BLK !
13 V$*EMT ( or )
14 ( V$@ $DECRYPT XCOUNT *TYPE )
15 VRTREC ;
```

Screen: 113

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 114

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Screen: 115

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 118

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 116

```
0 ( For demos: UMOVE $! )  
1  
2 '( $! ;S )(. )  
3  
4 : UMOVE ( a a n -- )  
5 (ROT OVER OVER U(  
6 IF  
7 ROT (CMOVE  
8 ELSE  
9 ROT CMOVE  
10 ENDIF ;  
11  
12 : $!  
13 OVER C@ 1+ UMOVE ;  
14  
15 -->
```

Screen: 119

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 117

```
0 ( For demos: ["] " )  
1  
2 : (") ( -- $ )  
3 R DUP C@ 1+ R) + >R ;  
4  
5 : "  
6 34 ( Ascii quote )  
7 STATE @  
8 IF ( cccc" -- )  
9 COMPILE (" ) WORD  
10 HERE C@ 1+ ALLOT  
11 ELSE  
12 WORD HERE ( cccc" -- $ )  
13 PAD $! PAD  
14 ENDIF ;  
15 IMMEDIATE
```

Screen: 120

```
0 ( X" ... " demo )  
1  
2 X" When you are going to take in  
3 hand any act, remind yourself w  
4 hat kind of an act it is. If yo  
5 u are going to bathe, place befo  
6 re yourself what happens in the  
7 bath: some splashing the water,  
8 others pushing against one anot  
9 her, others abusing one another,  
10 and some stealing: and thus wi  
11 th more safety you will undertak  
12 e the matter, if you say to your  
13 self, I now intend to bathe, and  
14 to maintain my will in a manner  
15 comformable to nature. And so
```


Screen: 121

```
0 you will do in every act: for t
1 hus if any hindrance to bathing
2 shall happen, let this thought b
3 e ready: it was not this only t
4 hat I intended, but I intended a
5 lso to maintain my will in a way
6 conformable to nature; but I sh
7 al not maintain it so, if I am v
8 exed at what happens.→→Epictetus
9 , translated by George Long, 187
10 7.→"
11
12 CR
13 ." The X-quote string is loaded"
14 CR
15
```

Screen: 122

```
0 ( V" ... " M: message-name demo)
1
2 V" There is an inconvenience whi
3 ch attends all abstruse reasonin
4 g, that it may silence, without
5 convincing an antagonist, and re
6 quires the same intense study to
7 make us sensible of its force,
8 that was at first requisite for
9 its invention. When we leave ou
10 r closet, and engage in the comm
11 on affairs of life, its conclusi
12 ons seem to vanish, like the pha
13 ntoms of the night on the appear
14 ance of the morning; and 'tis di
15 fficult for us to retain even th
```

Screen: 123

```
0 at conviction, which we had atta
1 in'd with difficulty. This is s
2 till more conspicuous in a long
3 chain of reasoning, where we mus
4 t preserve to the end the eviden
5 ce of the first propositions, an
6 d where we often lose sight of a
7 ll the most receiv'd maxims, eit
8 her of philosophy or common life
9 . I am not, however, without h
10 opes...→→David Hume, 1793.→"
11
12 M: MSGDEM1
13 CR
14 ." MSGDEM1 now exists."
15 CR
```

Screen: 124

```
0 ( X" ... " MSG: msg-name demo )
1
2 80 ALTINIT
3
4 X"
5 'Accessory No. 5 is a pocket com
6 pass and is used in connections
7 with putting. Like suppose for
8 inst. you land on the green abou
9 t 10 ft. from the cup, why the n
10 ext thing is to find out what di
11 rection the hole is at and this
12 can't be done and done right wit
13 hout a compass.→→ At lease I hav
14 e seen a whole lot of golfers tr
15 y and putt without no compass, a
```

Screen: 125

```
0 nd their ball has went from 10 t
1 o 45 ft. degrees to the right or
2 left of where the hole is actua
3 lly located. This is because th
4 ey was just guessing where as wi
5 th a compass they's no guess wor
6 k about it. If you miss a putt
7 with a compass to tell you just
8 where a hole is at, why it's bec
9 ause you can't putt so good.'→→R
10 ing Lardner on New Golf Accesori
11 es, 1924.→"
12
13 MSG: MSGDEM2
14 CR ." MSGDEM2 now exists." CR
15
```

Screen: 126

```
0 ( More MSG:'s )
1
2 X" The rat the cat I bought ca
3 ught escaped.→" MSG: M0
4
5 X" There are gold coins here!→"
6 MSG: M1
7
8 X" Aww, gee, Beave!→"
9 MSG: M2
10
11 X" You see, Watson, but you do n
12 ot observe.→" MSG: M3
13
14 X" Never look back; something ma
15 y be gaining on you.→" -->
```

Screen: 127

0 (More MSG:'s)
1
2 MSG: M4
3
4 X" 'The precise date at which th
5 e reversion to cap and gown took
6 place, as well as the fact that
7 it affected so large a number o
8 f schools at about the same time
9 , seems to have been due in some
10 measure to a wave of atavistic
11 sense of comformity and reputabi
12 lity that passed over the commun
13 ity at that period.'→→Thorstein
14 Veblen, 1899.→" MSG: M5
15

Screen: 130

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 128

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 131

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 129

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 132

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen 133 through 161 are blank.

Screen: 162

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 165

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 163

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 166

0 (Load Chain)
1
2 '(QUAN)(10 LOAD)
3 16 LOAD (utilities)
4
5 (Do not modify these lines)
6
7
8
9
10
11
12
13
14
15 -->

Screen: 164

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 167

0 (Load Chain, options screen)
1 28 LOAD (af, do not modify)
2 (34 LOAD (rgt & ctr justify)
3 (38 LOAD (fill justify)
4 (50 LOAD (coloring)
5 (52 LOAD (capitalization)
6 (54 LOAD (inverse video)
7 42 LOAD (af, do not modify)
8 (60 LOAD (text compression)
9 (68 LOAD (fmt'd num. output)
10 (70 LOAD (typed output)
11 (86 LOAD (B&W window output)
12 (88 LOAD (Color wndw output)
13 (select) = 1 of above 3)
14 (100 LOAD (Virtual mem. text)
15 (116 LOAD (" for demos)

Screen: 168

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 171

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 169

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 172

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 170

0 CONTENTS OF THIS DISK:
1
2 LOAD-CHAIN 166 LOAD
3
4 EFFICIENT (BUILDS DOES)
5 (ALSO LOADED BY TXTCMP) 56 LOAD
6 QUAN STRUCTURES 10 LOAD
7 TRANSIENT STRUCTURES 4 LOAD
8
9
10
11
12
13
14
15

Screen: 173

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 174

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

h12-0000
001 1 1700

001 1700

001 1700

001 1700

001 1700

Screen: 175

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

001 1700 201 1700

001 1700

001 1700

001 1700

001 1700

001 1700

Screen: 176

0 (Error messages)
1
2 Stack empty
3
4 Dictionary full
5
6 Wrong addressing mode
7
8 Is not unique
9
10 Value error
11
12 Disk address error
13
14 Stack full
15

Screen: 177

0 Disk Error!
1
2 Dictionary too big
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 178

0 (Error messages)
1
2 Use only in Definitions
3
4 Execution only
5
6 Conditionals not paired
7
8 Definition not finished
9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15

Screen: 179

0 Declare VOCABULARY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15